

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

**An Asynchronous Client-Side
Event Logger Model**

*Vagner Figuerêdo de Santana
Maria Cecília Calani Baranauskas*

Technical Report - IC-08-028 - Relatório Técnico

October - 2008 - Outubro

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

An Asynchronous Client-Side Event Logger Model

Vagner Figuerêdo de Santana* Maria Cecília Calani Baranauskas

Abstract

Web Usage Mining (WUM) usually considers server logs as a data source for collecting patterns of usage data. This solution presents limitations when the goal is to represent how users interact with specific user interface elements, since this approach may not have detailed information about what the users do in a Web page. This technical report presents a model for logging client-side events and an implementation of it in a tool called WELFIT (Web Event Logger and Flow Identification Tool). By using the model presented here miner systems can capture more detailed Web usage data, making possible a more fine-grained examination of Web pages usage. In addition, the model can help HCI (Human-Computer Interaction) practitioners to log client-side events of mobile devices, set-top boxes, Web pages, among other artifacts.

1 Introduction

Several studies have been addressed Web usage, ranging from Usability and Accessibility (A&U) guidelines to tools that analyze code, content, or logs of websites. Data Mining has been defined as the “analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner” [10]. Websites logs are commonly used as data source by Data Miners that focus on studying Web usage. The data mining of Web usage emerged as a new domain called Web Usage Mining (WUM) which is “the process of discovering and interpreting patterns of user access to the Web information systems by mining the data collected from user interactions with the system” [19].

WUM algorithms and tools focus mainly on server logs. Server-side data makes possible to identify the user route in a website requiring less effort, since Web server logs are a natural product of its use. However, server-side logs do not contain representative data about the interactions between the user and the Web page [8]. Client-side data have more detailed information about user actions in a Web page, but require more effort of the system to capture and transfer data to a local where researchers could analyze them.

Nowadays, HCI community count on tools that keep track of users behavior using mouse tracks (e.g., MouseTrack [2]) and eye tracks (e.g., eyebox2 [20]), but literature lacks studies focusing on data captured automatically from the whole diversity of users. Accessibility evaluation tools need to address some aspects usually not covered by evaluation tools based

*Instituto de Computação, Universidade Estadual de Campinas. Pesquisa apoiada pela CAPES através de bolsa de mestrado.

on mouse events or interaction by using a visual display. How tools that keep track of mouse or user's eye movements would help in an evaluation of a University's website that has teachers and students that are screen readers users?

In this context, we present a model to log event data that gets as many different events as possible, since with a large vocabulary of events in the logged data, researchers could perform many different analyses. One application of this model is to replay what clusters of users remotely do; this type of result would be costly to do with the videos used in tests in controlled environments.

The model requires the acceptance of the user to participate of the evaluation; the events recording are started as soon as s/he accepts it. In addition, a policy of privacy is implemented to allow the control of the type of events triggered by the participant that will be logged and transferred (e.g., if the "x" key is pressed then the data recorded will inform only that some key was pressed in some time and context of use).

After defining the model, we present an implementation of it as a case of study. The model and its implementation are part of the WELFIT (Web Event Logger and Flow Identification Tool) project, a work in progress tool to identify barriers that screen reader users face during a visit to a website being evaluated.

This report is organized as follows: the next section presents works related to client-side events capture; section 3 details the presented model; section 4 discusses implementation issues and details of this model in the Web context; finally, section 5 presents conclusions and future works.

2 Client-Side Event loggers

User Interface events are natural results of using windows based interfaces and their components (e.g., mouse movements, key strokes, mouse clicks, list selection, etc) [11]. Event logs produce results as frequency of use of certain functions, places where users spend more time and the sequence that users complete their tasks [22]. Since it is possible to record these events and they indicate the user behavior during the interface usage, they represent an important source of information regarding usability.

In this section, we will present WET and WebRemUSINE, client-side event loggers of Web pages, and discuss their common characteristics. Both tools use empirical data and are Web-based; however, their models can be generalized to log other kinds of client-side data logger. WET focuses in logging data during formal tests (i.e., in test controlled environments). In contrast, WebRemUSINE can be used in remote tests and in both formal and informal environments.

The evaluation and comparison of these Web-based tools, including the implementation of the model proposed here, are made to keep a theoretical unit and to exemplify how client-side logging can help evaluators to perform studies in different kinds of devices. Some of implementation issues are tightly closed to Web context, but the overall idea and model can be applied to other devices (e.g., mobile devices, set-top boxes, and video games).

Etgen and Cantor developed an event capture tool called WET (Web Event-Logging Tool). This tool, in contrast to traditional techniques of logging test sessions that use manual

record of user interactions, makes automatic capture of events that occur on the client-side, avoiding high costs of time and money present in manual data capture methods [8].

The log captured by WET is recorded in text format at client-side (i.e., in cookies). The available space in cookies is about 4 kilobytes. The capture starts when the user indicates the beginning of a task through clicking on the start option, which is visible in the pages that use the tool. The capture is interrupted when the user selects the stop option. The transmission of information to the server occurs only when the stop is triggered.

WET authors comment that future directions of the project involve the use of Java applet for logs transmission and the construction of a wizard to assist in the configuration of the tool [8]. This fact suggests that the aspects of WET that need improvement in dealing with the client-side logging are: some way to record more data, use a bigger event vocabulary, and do not depend on user actions.

WebRemUSINE [16] is a tool that makes automatic analysis of websites interaction logs in order to detect usability problems through remote evaluation. The analysis of records from WebRemUSINE is based on the comparison between the paths made by users and the optimum task model configured [16].

To capture events the tool makes use of a technique similar to the WET, but a wider range of events is used. The storage and transmission of the logs is done through a Java applet, which allowed the tool to avoid the storage capacity of cookies, scenario that may occur in WET in cases of longer sessions [16].

For the user, using this tool involves splitting its screen into two regions, one for the list of tasks that the participant must choose before starting each task, and the other containing the website being evaluated. Once the preparation phase is completed, the WebRemUSINE allows the number of sessions to grow without requiring more effort from specialists. For the authors, one of the improvements planned for the system is the automatic generation of task models of websites [16]. This fact suggests that the points of WebRemUSINE that need improvement in dealing to client-side logging are related to: not depend on pluggins installed on participant's device, not interfere in a significant way with the user interface being evaluated, and not depend on user actions.

These different approaches still have interesting challenges to reach their goals. The reader may note that the functions these two tools have in common result in a blueprint of what a client-side data-logger should do. In short, it must record client-side events that usually result in a large amount of data, interact with the participant of the test showing the status of the tool, and transmit the logged data to an eventual server. This was the starting point for the model specified in the next section. The model proposed in this technical report details other phases than just recording and transmitting, addressing other needs, and avoiding some of the limitations of the previous tools. More information regarding the well succeeded techniques, gaps, and limitations of websites client-side logger tools are found at [6].

3 The Client-Side Event Logger Model

This section describes the structure and components of the proposed model. It was built to address the requirements of websites evaluation tools (Table 1) elicited in a study of client-side loggers [6]. The requirements are organized in the Semiotic Ladder (SL), an artifact of Organizational Semiotics [21] which provides a framework for analysis of information systems under six different layers, considering the Information Technology Platform (i.e., physical, empirical, and syntactic layers) and the Human Information Functions as well (i.e., semantic, pragmatic, and social layers).

Table 1: Requirements for websites evaluation tools instantiated in the Semiotic Ladder [6].

Human Information Functions	
Social	Focus on the integration of accessibility and usability for the target audience of the evaluated website. Enabling remote testing during real use of the evaluated website. Interfere with the Web page as minimum as possible.
Pragmatic	Providing controls representing the status of the tool and user context during the test session. The operation of the tool should use two actions: one to start the capture, which can stay valid for more than one session, another to interrupt the capture, which may occur at any time.
Semantic	Providing high levels of abstraction without depending on task models, grammars, or specific events.
Information Technology Platform	
Syntactic	Using all available data (e.g., client-side events and server-side logs) in order to obtain correlations between them. The combination of the available data in different components can reveal information impossible to obtain in independent evaluations.
Empirical	Preventing that while processing or transmitting logs interfere with the use of evaluated interface. The tool should implement safe and effective techniques without impacting on the website usage.
Physical	Do not depend on resources or specific configuration of the participants devices (e.g., disk space, processor frequency, bandwidth, etc). The evaluation tool should include mechanisms to achieve their goals in different configurations of hardware and software.

The model was designed so that its set up and use require just one change in evaluated applications: a call to the client-side event logger. Thus, as soon as a participant starts the test session and accepts to participate of the test, the tool starts to record events occurred at the client-side until the participant cancels his/her participation.

The developed model started from the use case specification, the domain model elaboration, and responsibility assignment to conceptual classes. The model specification was designed from the refinement of the main goal of the tool: to *capture event at client-side and transmit the logged data to a server, where all analysis is made*. This

statement already indicates two components of the model: the **DataLogger**, responsible for capturing event data, and, the **Communicator**, responsible for transmitting logs to the server.

User interface events have a duration ranging from 10 milliseconds to one second [11], evidencing that the number of events that occurs during few minutes can be huge. Then, any data-logger that must transmit logged data through an information channel should have to compact the data to fill the requirements mentioned before. This brought the need for a component to compact all the data, the **LogCompactor**.

To deal with the amount of data recorded we propose the use of asynchronous communication with the server as a strategy to avoid using significant resources (i.e., disk and memory spaces), and to interfere as minimum as possible with the use of the website being evaluated. However, this solution can represent a limitation since it relies on the client bandwidth connection to the server. We will discuss how critic this point is in section 5, when we present results of this model.

To manipulate data and perform record, read, and remove functions, we used a **DataAccessObject**, responsible to access the data recorded in the users' device like logs and session information. In addition, we needed a way to interact with the user and show the status of the logger, responsibility of the **Facade** component. To address privacy policies we created a component responsible to check if the captured data can or cannot be sent to the server, the **PrivacyFilter**. Finally, we defined a **Factory** to create and build all the components together.

The following sections detail the function and main methods of each of these components; an overview of the tool structure is represented in Figure 1.

The model addresses the following requirements presented in Table 1:

- **Physical layer** requirement, since it is lightweight (see section 4 for implementation details) and depends on few resources of the users' devices, achieving its goal in different configurations of hardware and software;
- **Empirical layer** requirement, since it process and transmits logs without interfering with the use of the evaluated interface;
- **Syntactic layer** requirement, since it uses all event available data that do not impact on security problems;
- The **Semantic layer** requirement is partially addressed. The model aims to log usage data without depending on specific task models, grammars, or events, but it does not foresee the analysis of the data to reach high levels of abstraction;
- **Pragmatic layer** requirement, since it provides controls representing the status of the tool and user context during the user interface test session, which makes possible that the user interrupts the capture at any time;
- **Social layer** requirement, since it focuses on interfering as minimum as possible with the user interface being evaluated. In addition, it makes possible the integration of A&U, since it keeps track of the interaction through different devices.

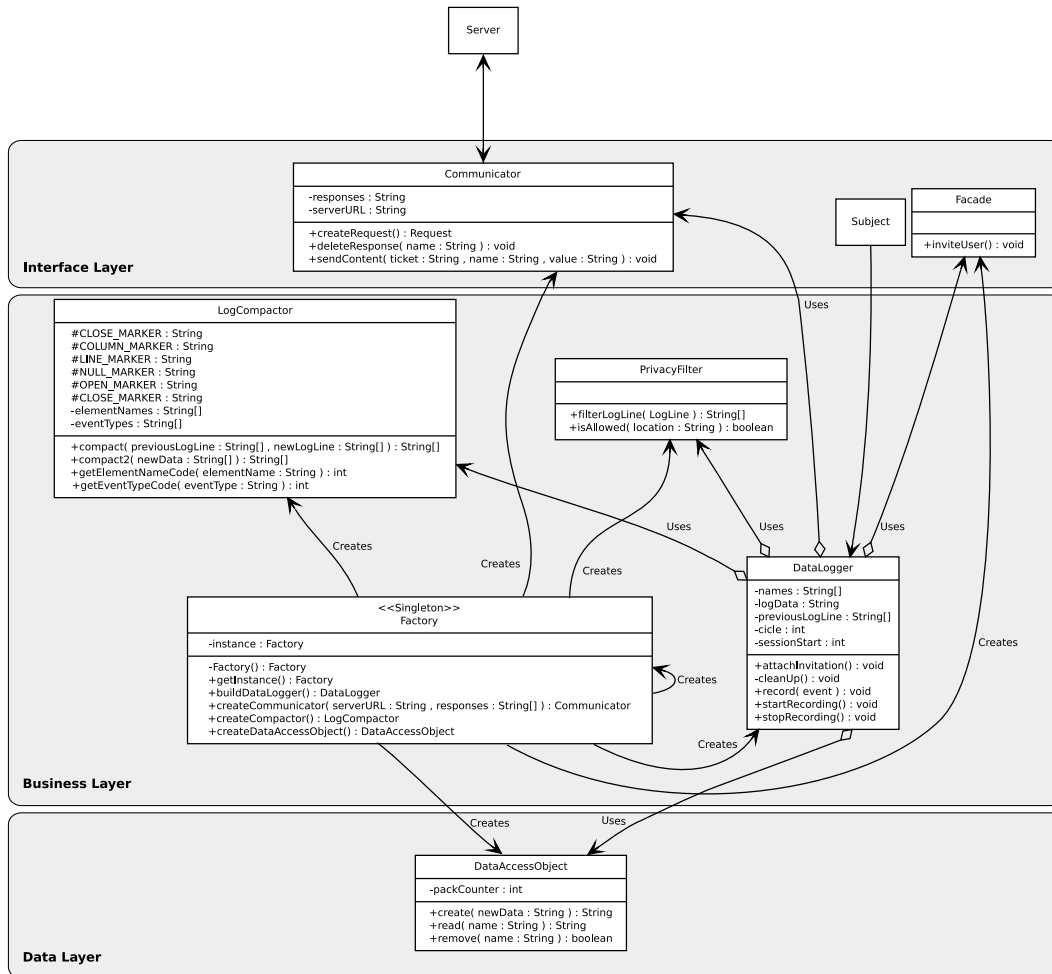


Figure 1: The overview of the client-side event logger model. This representation tries to simplify the whole design so it suppress some attributes, methods (e.g., setters, getters), and some data and abstract classes.

The following sections detail the components of the model: **DataLogger**, **Communicator**, **LogCompactor**, **DataAccessObject**, **Facade**, and **PrivacyFilter**.

3.1 DataLogger

The **DataLogger** is the central component of the model. The class contains the logic to manage the logging of events. It communicates with other classes from data layer (e.g., **DataAccessObject**) and interface layer (e.g., **Facade**, **Communicator**). Initially, the **DataLogger** is attached to the subject to be observed (e.g., Window object), so it can be notified to record all the events occurred. It is inspired on the GoF (Gang of Four) Observer Pattern. Observer is a behavioral pattern that defines a way to notify dependent classes when some change occurs in the subject class [9].

The main attributes of this component are:

- *Log data* - contains the data of the current package being built. As soon as the logged data reaches a defined size, the **DataLogger** creates a package to be sent to the server;
- *Package size* - define the limit in bytes of the packages of logs;
- *Previous log line* - used to check what information changed between the previous event and the new one. It is used in the compaction procedure, detailed in **LogCompactor** section;
- *Clean up cycle* - define the interval in seconds in which the packages already confirmed by the server are deleted;
- *Session start, IP address, a ticket, and other information* - used to identify users' sessions and events stream, since the IP of different participations can be the same (e.g., when requests come from an corporative network).

As soon as it starts to run, **DataLogger** asks the **Facade** to show the invitation to the user. Then, if the invitation is accepted, the **DataLogger** is notified to start the event capture, when it attaches the record function to all event handlers of the subject object.

The **DataLogger** can also be stopped from recording events in two ways: first, when the user activates the button to stop recording; second, due to some problem occurred at the client-side (e.g., if the number of accumulated packages reaches a defined limit). It is also important to note that every change occurred in the status of the tool must be informed to the user.

While the participant interacts with the subject (e.g., window-based component), the events are triggered. Thus, for each event triggered the **DataLogger** records all information and build the log line. This line is passed to **PrivacyFilter**, further passed to the **LogCompactor**, and then, finally, added to the current package.

The **DataLogger** component looks for every defined event, since capturing all client-side events provide researchers ways of finding different patterns, a requirement to be addressed by the model.

As log lines are added to the current package the **DataLogger** verify if it has reached the defined package size. If it is the case, then the **DataLogger** asks the **Communicator** to send the package to the server. Additionally, it calls the **DataAccessObject** to record the package just sent. This register is used if the **DataLogger** needs to send it again to the server.

Finally, at a define cycle; the **DataLogger** executes a method to verify if the server has sent a confirmation for the already sent packages. Thus, for every confirmation, the **DataLogger** asks the **DataAccessObject** to delete packages confirmed by the server and asks the **Communicator** to delete the confirmation messages that were already used. When cleaning up, if the **DataLogger** finds an error message or does not find a confirmation message, then the **DataLogger** calls the **Communicator** to send the respective packages again.

3.2 Communicator

The **Communicator** component is responsible for controlling the transmission of the logged data to the server. It keeps all the information regarding the identification of packages being sent and keeps all the responses emitted by the server. Additionally, it uses asynchronous transient communications to the server (i.e., by asynchronous we mean that the client sends a package to the server asynchronously while participants are using the Web pages; by transient we mean that the server is already waiting for requests). This characteristic is one of the improvements over the previous mentioned tools, enabling the recording of a wide variety of events without interfering with the website usage.

The **Communicator** must use secure methods to transfer the logged data. The packages can be ciphered or transferred through a secure channel, ensuring that other programs cannot intercept the packages. However, due to resources dealing with processing time or technologies, developers can leave the data less protected if they configure a less restrictive **PrivacyFilter**.

The main attributes of this component are:

- *Responses* - a collection of all responses sent by the server;
- *Server URL* - the address to where all packages are sent.

The **Communicator** initiates by establishing a connection with the server pointed by the server URL attribute. Then, for each package mounted by **DataLogger**, the **Communicator** creates a request to the server sending the package just built. Each package is identified by a ticket, a package name, and a value.

As soon as it receives the server response, it records the response that will be used when the **DataLogger** run the clean up procedure. Additionally, it also retrieves and deletes server responses as needed. These cases occur when the **DataLogger** cleans up, since it reads all messages and deletes confirmation messages that will not be used anymore.

Table 2: Event flow data format example using codes 1 and 2 to events *mousemove* and *click*, respectively, and codes 10 and 11 to elements [*object HTMLBodyElement*] and [*object HTMLImageElement*], respectively.

Timestamp(ms)	Event code	X coordinate	Y coordinate	Element code
400	1	50	50	10
500	1	60	60	10
600	1	70	70	10
700	1	80	80	11
800	2	90	90	11
900	1	100	100	11
1000	1	110	110	11
1100	2	120	120	11

3.3 LogCompactor

The **LogCompactor** was introduced to avoid the heavy consume of client’s bandwidth connection, that may occur if the raw log is transferred to the server. Additionally, the compact technique to be used needs to be lightweight, so that the required processing is not noticed by users while they are using the artifact in evaluation.

In order to define the compaction technique to use, we reduced the search scope to the simple (and lightweight) compaction techniques. We tried using frequency table based compaction, in which we represent the most frequent values with short length codes. We got a compaction around 35%. But when analyzing the raw and the compacted data we found that the codes were commonly present in subsequent log lines. The Table 2 illustrates the sequence found in logged data in which codes appear in subsequent log lines.

After that analysis and due to event flow data format, we decide to experiment representing only the changed values (i.e., if an event is repeated many times at the same place, we just record the complete information of the first occurrence and for the next ones we record only the time they occurred, since the other values haven’t changed). For an example, see Table 3. This technique brings compaction around 70% without making the compacting procedure a processor-consuming task.

Before detailing how the compactor works it is important to state that the log lines we use have fixed column length, even if the just recorded event does not have all the properties the data-logger is looking for. This approach was used to prevent the client to send control data to the server (e.g., property identification). Additionally, due to events flow data format and the compaction used, we prevent the null marker to represent null/undefined values in log lines, since they come as empty values until they change into event property data (see Table 3).

The compaction has two steps:

1. Since the lines are fixed column length, the first step of compaction is a loop that verify, for each log line value, if new log line value is equal to the respective one in

Table 3: An example of how 4 events can be compacted representing only data that changed. First, the participant moved the mouse over the logo image from coordinates 50 to 100 on the X-axis, then he clicked over the logo at 600 ms after the session has started, and then after a second he clicked again over the logo. The last column shows how null values are represented to maintain the fixed column length representation.

Time-stamp (ms)	Event name	X coordinate	Y coordinate	Target ID	Element name	Referrer
400	mouseover	50	200	logo	[object HTMLImageElement]	null
500		100				
600	click					
1600						

the previous log line. If this is the case, then this position on the compacted log line is replaced by an empty string. Finally, at the end of the loop the compacted log line is returned to the caller component;

2. The second step was added to improve the compaction obtained by the first step and reduce the overhead of markers used to separate empty values of the fixed length log lines. For this purpose, we defined a markup to replace repetitions of column separators (e.g., commas) by markers indicating the number of repetitions of the separator. Note that this will be done only if it is worth it (i.e., the number of commas is greater than the sum of repetition markers and the number used to represent the repetitions). An example of compaction of the Table 3 log data would be:

```
400,mouseover,50,200,logo,[object HTMLImageElement],-@
500,,100{3}@
600,click{5}@
1600{6}
```

Where comma (,) is the column marker, the hyphen (-) is the null marker, the at (@) is the line marker, and braces ({,}) are open and close markers. We should note that all the content must be escaped depending on the implemented markup preventing conflicts between markers and event data. This means that all markers used to represent column (,), end of line (@), and so on, that appear in the logged data must be concatenated with another marker to avoid ambiguity between markup and data. For example, using the back slash to escape column and line markers we would get the “\,” and “\@”.

Complementing this approach, we defined a fixed codification to event types and to element names, reducing values as mousemove and click to 1 and 2, for example. In addition, we used the bigger base available to represent number values with less bytes (e.g.,

hexadecimal values, base 32, and so on). The following lines detail these techniques using the raw log data from Table 3:

1. Raw log data using markers to represent column, line, and null values (211 bytes):

```
400,mouseover,50,200,logo,[object HTMLImageElement],-@
500,mouseover,100,200,logo,[object HTMLImageElement],-@
600,click,100,200,logo,[object HTMLImageElement],-@
1600,click,100,200,logo,[object HTMLImageElement],-
```

2. Log data using event type codes, element name codes, and base 32 to represent numeric values (80 bytes, 62% of compaction):

```
cg,2,1i,68,logo,1,-@
fk,2,34,68,logo,1,-@
io,1,34,68,logo,1,-@
lio,1,34,68,logo,1,-
```

3. A first step compaction (50 bytes, 76% of compaction):

```
cg,2,1i,68,logo,1,-@
fk,,34,,,,@
io,1,,,,,@
lio,,,,,
```

4. A second step compaction (44 bytes, 79% of compaction):

```
cg,2,1i,68,logo,1,-@
fk,,34{4}@
io,1{5}@
lio{6}
```

3.4 DataAccessObject

The **DataAccessObject** is based on the Data Access Object (DAO) J2EE Design Pattern. DAO provides a solution to abstract and encapsulate the access to the persistent storage, managing the connection with the data source to retrieve and record data [1]. The **DataAccessObject** component is located at data layer and its responsibility is to perform record, read, and remove actions as the **DataLogger** requests it. **DataAccessObject** is the component that access and retrieve data recorded in the users' device like logs and session information. In addition, the component defines a name for each package when creating registries.

3.5 Facade

The **Facade** is the user interface of the logger. It is inspired on the GoF (Gang of Four) Facade Pattern. Facade is a structural pattern that “defines a higher-level interface that makes the subsystem easier to use” [9].

The **Facade** is the class that shows the status of the tool and offer controls to the participant interact with the system (i.e., **DataLogger**). Initially, the **Facade** is asked by **DataLogger** to show the invitation and keep the participants aware of what is happening with the tool. Then, each activation of **Facade**’s controls calls the respective method at **DataLogger** component (e.g., stop recording).

3.6 PrivacyFilter

The **PrivacyFilter** is the component that uses previously defined policies (e.g., not record which key is pressed when a key press event occur, not record events occurred in a certain location). These policies can be defined in two levels: first level rules with priority are defined by the administrator of the tool and second level rules can be defined by the administrator of the application being evaluated, if they do not conflict with the former.

Two methods are used to define the rules while the tool is recording events:

1. Administrators can specify if some locations cannot be logged through the *isAllowed* method. Thus, when a participant accesses some location specified in some rule, then the tool does not record events in that location (e.g., login.html). Before recording, the **DataLogger** asks permission for **PrivacyFilter** using the method *isAllowed*;
2. Administrators can specify which information of events cannot be logged through the *filterLogLine* method. With this method it is possible to define policies for each type of event. For example, if there’s a rule to filter every key value when a participant presses a key, then when some key event is recorded then all values related to whichever keys were used will be filtered and, consequently, not sent to the server.

3.7 Factory

The **Factory** is the model creator class. The **Factory** contains the information to instantiate and build all components. First, the **Factory** instantiates itself, following the Singleton GoF pattern. Singleton is a creational pattern that provides a global point of access to it [9]. In addition, it uses two other GoF creational patterns: Factory Method, that “lets a class defer instantiation to subclasses” [9]; and Builder, that “separates the construction of complex object from its representation so that the same construction process can create different representations” [9].

This component was designed this way to allow the model to be extended allowing the definition of other classes used to assembly the **DataLogger**.

4 Implementation issues and results

This section discusses some of the implementation issues and results we achieved when applying the presented model. As mentioned before, this work is part of a project involving the evaluation of websites. Thus, the implementation was made using JavaScript, a script language that is native of the newest Web browsers.

The use of this implementation requires only that websites being evaluated insert a reference to the JavaScript in the header of Web pages. From that point, each time an user access the page, the server fills up the script with information unavailable from JavaScript (e.g., client's IP, a global identifier for that session, etc.) before serving it. Then, as soon as the script is loaded at client's browser, the tool starts to run.

This model behaved well in the implementation using JavaScript. However, some components faced limitations in dealing with security constraints. The main issues are related to the space available to record information on client's device and to transfer the data to a different domain from the website being evaluated.

The space available to record information in the client's device is restricted. The first solution was to use cookies, but the first negative point is that they are limited to a size of 4 kilobytes (kB) and each domain can specify only 20 cookies [15]. This results in a space of 80kB to a tool that captures approximately 4kB of raw log per second. This might be a problem, but the asynchronicity of the model dealt with that. Afterwards, the problem was the time required to record, retrieve, and delete the packages without interfering with the use of the website. Initially we used the same limits of cookies. Then, to avoid the time required to manipulate cookies we used the Web page structure in memory, also known as Document Object Model (DOM) tree. Finally, we used application cookies only to deal with error recovery and to maintain information valid for more than one session (e.g., the acceptance of the user to participate of the evaluation).

Another point that contributed to the solution of keeping the logged data in DOM tree is that some users deactivate cookie support, so the direct dependency of cookies might reduce the amount of cases to study. We used the same limits imposed by cookies the keep the tool as lightweight as possible, as stated in the requirements.

The data unit we are saving in cookies are the packages of log, then we are limited to the use of 20 packages in the DOM tree. Thus, either at clean up cycle or at reaching 10 accumulated packages the **DataLogger** tries to send it all again. If many errors occur and the limit of 20 accumulated packages in the client-device is reached, then the tool **DataLogger** stops to record events. This solution to persist data showed to be efficient and effective.

The bigger issue implementing the proposed model in JavaScript was the asynchronous cross-domain transmissions. The problem was to deal with security restrictions of the XMLHttpRequest, a JavaScript object widely used object in AJAX (Asynchronous JavaScript And XML) applications, which just allows connection between Web pages/applications hosted at the same domain.

The security restriction is called Same Origin Policy, it "prevents document or script loaded from one origin from getting or setting properties of a document from a different origin" [18]. Then, if you try to make an XMLHttpRequest from one domain to another,

your Web browser probably will pop up a permission denied message.

At first sight, the Same Origin Policy may seem too restrictive, since it blocks the use of Web services directly via XMLHttpRequests. However, according to Levitt [12], allowing scripts to access any domain from within a Web page opens up users to potential exploitation. Next we discuss some solutions to deal with this policy and finally comment about the solution we have chosen for this case study.

One of the best initiatives we found is the use of Signed Scripts. This involves “generating a digital signature and associating that signature with the script it signs” [17]. With this signature the scripts can obtain privileges to, for example, use cross-domain XMLHttpRequests. However, it is an initiative of the Mozilla project and depends on the security model implemented in Mozilla’s compatible Web browsers, leaving aside all users of the other browsers, a price we do not want to pay.

Another interesting solution we found is to use server-side proxy programs, since the majority of server-side programming languages allow the cross domains connections. Thus, a participant viewing the Web page `index.html` at domain **www.example.com** would trigger events that will first be sent to the **www.example.com** Web server, and then a proxy program would dispatch the information to the location of the server of the tool (e.g., **www.toollocation.com**). This is indeed effective, but this solution came with two negative effects. First, all websites administrators using the logger would have to configure the proxy, a task that could be simple but also can even require the download of modules or installation of complements. Second, the Web server of the website using the tool would face a notorious overhead, since each package being sent to the tool server would be passed first to the **www.example.com** Web server. This price, web site administrators may not pay.

A very simple solution is the Iframe Proxy. In this solution, the information is exchanged by the URL of an iframe HTML element. This is possible because the document containing the iframe and the document being referenced in it can both update the iframe [7]. Thus, the **www.example.com** application can put a message in the URL of the iframe calling the **www.toollocation.com** and then the referenced page can take this message, process it, and put the response back into the iframe’s URL. After that, **www.example.com** can use it at some timeout-defined cycle. This solution is interesting, however, as the iframe is updated some Web browsers (e.g., Microsoft’s Internet Explorer) plays a sound, a price users should not have to pay.

Another client-side solution is the use of a Flash Proxy. Due to the policy used in movies in Flash to exchange data between domains it can be used as a proxy to send data to another server [14]. However the dependency on Flash pluggins and pluggins’ versions would difficult the use of the logger, conflicting with the requirements presented before.

Some proposed solutions to the Same Origin Policy are: XMLHttpRequest, module tag, content restriction header, W3C Access Control List (ACL) System, and Flash’s `crossdomain.xml` [3]. However, the proposed solutions that would give to JavaScript programmers the power to perform cross-domain requests are XMLHttpRequest and module tag. Thus, we will detail these two approaches.

Before presenting XMLHttpRequest, we need to introduce what is JSON (JavaScript Object Notation). JSON is a data interchange format based on a safe subset of JavaScript to

represent simple or complex structured data [4]. Therefore, “JSONRequest is proposed as a new browser service that allows data exchange with any JSON data server without exposing users or organization to harm” [4].

JSONRequest is a solution that acts like the XMLHttpRequest with the following changes [3]:

- Use a minimal set of HTTP headers, reducing the overall size of requests;
- Cookies would not be transferred, avoiding cross-site cookie issues;
- Accept only JSON text, ensuring that code could not be sent for execution;
- Each communication failure is randomly scheduled to a retry, frustrating certain attacks;
- Return a sequence number so that each response can be easily associated with the originating request;
- Support duplex connections, enabling server to initiate communications.

The <module> tag proposes to divide a Web page into a collection of modules that are secure from each other and provide safe communication. This solution is somewhat similar to the iframe proxy presented before, but it specifies and foresees different uses. The communication between the page and modules is only allowed using send/receive functions and allows exchange data only in JSON text format. In addition, modules can cooperate only with the Web page, increasing the level of security between different modules. Thus, due to control over the send/receive communication, crossing domain between the module and the Web page is irrelevant [5]. According to Crockford [5], the module tag intent is to begin the process of reaching a consensus on a new Web browser security model, since Web applications are significantly ahead of Web browsers technologies.

The solution we are using in this implementation is based on an approach presented in Levitt [13].

This approach uses the fact that when the script tag is loaded, Web browsers execute the loaded script, also known as dynamic script tag. Thus, manipulating the DOM tree and making requests through the creation of script tags dynamically allow asynchronous cross-domain communication.

When a script tag is created, an HTTP GET request is sent to the server module, which processes it and serves the response as JSON content to the caller Web page. HTTP GET method is used because this approach does not support HTTP POST method. JSON is used to avoid the parse of other formats (e.g., XML) since JSON can be used directly by JavaScripts programs, what in fact makes this implementation an AJAJ (Asynchronous JavaScript And JSON) application. Finally, the Web page can use the response and delete the respective script tag object from DOM tree.

Levitt [13] describes pros and cons of this approach compared to XMLHttpRequest. Some of them are:

- Dynamic script tag runs identically on more Web browsers than XMLHttpRequest. An example of that is the Microsoft’s Internet Explorer implementation of XMLHttpRequest, that it is different from other Web browsers and requires a compatibility layer or additional verifications;
- Dynamic script tag does not support HTTP POST method;
- Dynamic script tag cannot send and receive individual HTTP headers; consequently, it cannot handle errors gracefully through HTTP.

These points might suggest not using it. However, we are aiming not rely on plugins or any other components other than JavaScript and Web pages. The dependence on plugins brings the need for installing other software, configuring it, etc. This is a temporary solution since the technology of most popular Web browsers does not support completely the model we proposed. Additionally, the security issues due to transmission of coded logs through HTTP GET can be facilitated through restrictive configuration of **PrivacyFilter**, as presented before.

Tests we made in the homepage of the website **www.todosnos.unicamp.br**, using random mouse movements over links and page elements, showed that each second of interaction results in approximately 1kB of compacted logs. Therefore, any participant using a connection that supports the transmission of 1kB per second plus the mean of bandwidth connection used by the participant to surf the Web will allow the logger to behave accordingly to the design and does not interfere with the use of the website. If it is not the case then, as specified in the model, the tool must become inactive and does not record data if the available bandwidth does not allow the transmission of the logged data according to the defined requirements.

5 Conclusion and future works

The model presented in this technical report is an alternative to log client-side usage data. It can supply data to other applications focusing on discover patterns of what clusters of users do or just to replay user’s actions during tests in controlled environments, like the well known video tools used in usability tests. The implementation showed to be lightweight and addressed the requirements stated for a client-side event logger tool.

During implementation and use of this model, maintainers and developers must always keep security and privacy in mind, since the information being captured and transmitted can be critic if it is not made in a safe manner. Accordingly, users must always be aware of what is happening in their device and accept to participate before the logger starts to record events, since the free record of this kind of information without warning the user would characterize the tool as a spyware, for example.

Among the limitations of the model is the inability to register any action if the user agent does not support the language used to implement it (e.g., JavaScript). Moreover, if the device using a data-logger based in this model has a connection that transmits data in a lower speed than data is collected, then the data-logger will not work properly and consequently stop recording events.

There are different approaches for compacting data, but as stated before we aimed to find a technique that could balance effectiveness and efficiency for events flow log data. Thus, one future work to this model is to test different compacting techniques to obtain an improved lightweight compaction. Another topic to be studied is the addition of detection and/or correction mechanism for the transmission of compacted data.

Another future improvement is to change the dynamic script tag approach into a plugin independent more adequate approach, as soon as it gets implemented (e.g., JSONRequest, module tag).

Finally, the completion of the WELFIT project foresees the implementation of the server module, complementing the data-logger. This module will perform WUM using the logged data to identify barriers that participants faced when navigating through websites. This project aims to infer high-level patterns of usage without depending on specific task models or grammars. Reaching this goal will make WELFIT address the Semantic Layer requirement.

References

- [1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2nd edition, 2003.
- [2] E. Arroyo, T. Selker, and W. Wei. Usability tool for analysis of web designs using mouse tracks. In *Proceedings of ACM CHI 2006 Conference on Human Factors in Computing Systems*, volume 2 of *Work-in-progress*, pages 484–489, 2006.
- [3] B. Ashley. Shaping the future of secure ajax mashups, 2007.
- [4] D. Crockford. *Jsonrequest*, 2006.
- [5] D. Crockford. The `<module>` tag, 2006.
- [6] V. F. de Santana and M. C. C. Baranauskas. A prospect of websites evaluation tools based on event logs. In P. Forbrig, F. Paternò, and A. M. Pejtersen, editors, *Human-Computer Interaction Symposium*, volume 272 of *IFIP International Federation for Information Processing*, pages 99–104. Springer Boston, 2008.
- [7] Dojo Toolkit. Cross domain xmlhttprequest using an iframe proxy, 2006.
- [8] M. Etgen and J. Cantor. What does getting wet (web event-logging tool) mean for web usability? In *Proceedings of 5th Conference on Human Factors & the Web*, 1999.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.
- [10] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. The MIT Press, 2001.
- [11] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. *ACM Comput. Surv.*, 32(4):384–421, 2000.

- [12] J. Levitt. Fixing ajax: Xmlhttprequest considered harmful, 2005.
- [13] J. Levitt. Json and the dynamic script tag: Easy, xml-less web services for javascript, 2005.
- [14] J. Levitt. Flash to the rescue, 2006.
- [15] Netscape Communications Corporation. *Client-Side JavaScript Reference*. 1999.
- [16] L. Paganelli and F. Paternò. Intelligent analysis of user interactions with web applications. In *IUI '02: Proceedings of the 7th international conference on Intelligent user interfaces*, pages 111–118, New York, NY, USA, 2002. ACM.
- [17] J. Ruderman. Signed scripts in mozilla, 2007.
- [18] J. Ruderman. The same origin policy, 2008.
- [19] C. Shahabi and F. Banaei-Kashani. A framework for efficient and anonymous web usage mining based on client-side tracking. 2002.
- [20] D. Skeen. Eye-tracking device lets billboards know when you look at them, 2007.
- [21] R. Stamper. A semiotic theory of information and information systems / applied semiotics. In *Invited papers for the ICL/University of Newcastle Seminar on 'Information', September 6–10, 1993*, 1993.
- [22] D. Woo and J. Mori. Accessibility: A tool for usability evaluation. In M. Masoodian, S. Jones, and B. Rogers, editors, *Computer Human Interaction, 6th Asia Pacific Conference, APCHI 2004, Rotorua, New Zealand, June 29 - July 2, 2004, Proceedings*, volume 3101 of *Lecture Notes in Computer Science*, pages 531–539. Springer, 2004.