A Framework for Studying the Use of Wikis in Knowledge Work Using Client-Side Access Data

Uri Dekel
School of Computer Science, Carnegie Mellon University
udekel@cs.cmu.edu

Abstract

While measurements of wiki usage typically focus on the *active* contribution of content, information on the *passive* use of existing content can be valuable for a range of commercial and research purposes. In particular, such data is necessary for reconstructing the context or tracing the flow of information in settings where wikis are used as collaboration platforms in knowledge work that relies on specialized tools, such as software development.

Meeting these needs requires detailed knowledge of user behavior, such as the duration for which a page was read and the sections visible at each point. This data cannot be collected by present wiki implementations and must be collected from the client-side, which presents a range of technical and privacy problems. In addition, this data must be correlated with traces of interaction with other tools.

In this paper we present an approach for solving these problems in which scripts embedded by the wiki server are executed by the client browser, and report on the user's interaction with that document along with relevant structural information. These reports are relayed to a comprehensive framework for storing and accessing interaction and context data from the wiki and from additional tools used in knowledge work. This framework can be used to correlate these traces to obtain a complete view of the user's work across tools, or to approximate his context at specific points in time.

Categories and Subject Descriptors H.5.3 [*Group and Organization Interfaces*]: Computer-supported cooperative work, Web-based Interaction; H.5.4 [*Hypertext/Hypermedia*]: Navigation

General Terms Measurement, Human Factors

Keywords Wiki, Software Engineering, User Activity Tracking

1. Introduction

1.1 Active and passive usage data

The success of an online community is often measured by the level of *active participation* in the creation and editing of content. Much research is concerned with increasing such participation (e.g., [2]). Wikis are well poised for success in this arena as they lower the barriers to contribution by avoiding complex authorization and modification processes and doing away with difficult formatting notations.

In addition, wikis are different from most online communities in that contributions eventually form a comprehensive and structured body of knowledge in which each document can be perused as a cohesive unit without revealing traces of its creation process. To illustrate, it is common practice in most wikis to separate the content pages, relevant to most readers, from the discussions, relevant primarily to contributors. In contrast, while traditional communities such as mailing lists and bulletin boards are often divided into topics, knowledge consumers must typically read and distill information from an entire thread. Therefore, wiki contents are likely to attract a significantly larger audience that seeks immediate answers and is agnostic to the process by which the content was created.

Thus, we argue that important measures for the success of a wiki involve the *passive* use of its contents: what articles are accessed, for how long, and what the user does subsequently. We use the term *passive* because the information may be absorbed by a reader without leaving any obvious manifestations.

These issues are of obvious relevance to operators of for-profit wikis, who strive to measure exposure to their materials, and then maximize it by means such as collaborative filtering [7]. Details about user activities can also be valuable for profiling and data mining. Such wiki operators may also want to identify the parts of large articles that are visible at each point, to improve collaborative filtering, or to measure the actual exposure time to an ad.

1.2 Motivation

Our focus in this paper, however, is on the research implications of this passive knowledge with respect to knowledge work. With the rising prevalence of wikis as collaboration platforms in organizational settings and domains that rely on specialized tools, information on their passive use may play an important role in research on their integration in the organization, and in more general research on the flow of knowledge in this type of knowledge work. Such research may help improve the collaborative value of wikis in these domains.

One representative example, which has lead us to this topic and on which we focus in this paper, is collaborative software development. Programming activities traditionally take place in the integrated development environment (IDE), which offers a platform for creating and testing code. In collaborative settings, developers tend to rely on additional project-support tools that are often web-based, such as issue tracking systems. Wikis have gained a foothold in some projects as a platform for brainstorming, communicating, and documentation, and occasionally even replace traditional project-support tools.

Software development can be characterized as a series of design and implementation decisions that are manifested as actual software artifacts or as peripheral project artifacts. A major problem in software maintenance involves reproducing the decisions and the rationale behind existing artifacts [4]. Individual developers, and in some cases entire teams, often fail to capture the rationale explicitly [6]. An understanding of the artifacts involves an attempt to reconstruct the mental model of the developer and his assumptions at the time the decisions were made [5].

One factor in these reconstruction attempts is the context in which decisions were made. For example, what were the developer's immediate goals and of what information was he aware? While it might be impossible to accurately determine the answers to these questions, it may be possible to identify indirect evidence of things that could have been part of this context. For example, what material was visible at the time or was recently read?

To illustrate this issue and understand how records of passive access may be useful, let us consider the following hypothetical scenario. A software team is hired to develop a system for automating an organization's core business processes using agile techniques. The requirements engineer meets weekly with the customers, and captures the process as business rules in the wiki. During the week he adds to the wiki detailed scenarios and mock prototypes which he reviews in the subsequent meeting, possibly leading to changes in the requirements. Meanwhile, the rest of the development team tries to develop code that matches the preliminary business rules and detailed scenarios. They consult the wiki, and write the appropriate code. Some prototypes are later presented in the clients meeting.

It is common in software development for requirements to change after the code is written, and this presents significant challenges. Before the project is delivered, how can the team efficiently verify that the code corresponds to the latest requirements? Furthermore, how can the team be made aware that certain code fragments have become outdated as a result of a change to a requirement that they depends on?

Unless developers were careful to systematically document the dependencies on specific versions of specific business rules, it may be difficult to achieve the above goals.

However, suppose that we could capture a structured record of each developer's activities within the IDE, and of his activities in browsing the wiki. It is likely that in many cases, though not in all, the developer will study specific business rules in the wiki in temporal proximity to the time when relevant code is written, so that this information may give context to the code. If we correlate the two traces, the evidence can be used to reproduce these contexts. Given a code section, we could approximate which articles were examined at the times it was edited, and whether the versions have changed since then. One could also foresee an awareness-support system that periodically queries the wiki for all changed materials, identifies all code sections which were created in the context of earlier versions of these materials, and then notifies the developers.

Of course, the construction of a meaningful and accurate context requires us to look at a granularity that is smaller and more precise than an entire article, namely specific sections or even individual lines. For instance, if the developer has not viewed a particular business rule which was outside his viewport, then it is unlikely to have been part of the context, and may not be relevant even if it later changes. In addition, we must reduce the noisy effects of behaviors such as navigation mistakes, rapid skimming. This requires an elaborate record of wiki activities.

1.3 About this work

This paper is concerned with the problem of obtaining and applying the necessary data about passive usage of wikis to support research and collaborative work in domains such as software engineering, as described earlier. For reasons which we shall later describe, such information can only be collected from the side of the client browser, which presents significant technical and privacy challenges.

One contribution of our work is in proposing a novel technique, based on the use of embedded scripts, to capture this data. While others have concurrently applied similar technical principles to capture detailed recording of interaction with web pages for usability purposes, our tools are focused on large scale data analysis over multiple sessions and collect relevant wiki-specific data, including structural information. The major contribution and what primarily distinguishes this work, however, is a framework for representing and accumulating this data, which helps correlate it with interaction traces from other tools, such as the IDEs used in software development.

Outline: the rest of this paper is organized as follows: Sec. 2 describes the challenges in collecting passive usage data from wikis, our approach, and its implementation.

Sec. 3 describes our *eMoose* framework for representing and storing the collected data. We conclude and present our current research directions in Sec. 4.

2. Gathering Wiki usage data

We begin with the first problem, of gathering the usage data necessary for meeting the information needs described in the introduction. In our discussion, we shall use the term *session* to denote a user's interaction with a specific wiki page, which begins with the request and ends when the browser window is closed or when it is used for another request. A user's typical interaction with a wiki typically consists of several sessions, which can be concurrent if multiple windows or tabs are used.

2.1 Necessity for client-side data

Usage data that benefits the operator of a website is customarily derived from logs maintained by the site's server. Since wikis are web-based, it is typically possible to obtain a log of page requests, which can be used to track the number of sessions and perform rudimentary user profiling [7].

However, page requests are not necessarily indicative that the page was actually read. For example, the user may immediately close the page or move elsewhere, without making a subsequent request that would be recorded in the log. In addition, wiki pages often significantly exceed the available screen real-estate, allowing only a small portion to be visible at a time. While users may scroll and read the entire document, they may also seek specific information. Visual skimming, text-based searches in the browser, and internal links such as from a table of contents can all be used to bypass much of a page's contents and focus on specific information.

Clearly, a determination of the potentially influencing content can only be made after the page has been rendered on the client's browser, since we must track for how long it remains open. In addition, it may be necessary to aggregate the movements of the viewport over time in order to determine what material had a reasonable chance of being absorbed. Such aggregation may start as soon as the page is loaded and continue until the context is calculated.

Like all web pages, the specific rendering of a wiki page may depend on many factors such as the browser, the machine, the window size, the font size, etc. In order to use the information about viewport coordinates and determine what is actually visible, we must know the coordinates of relevant elements, such as subsections or embedded diagrams in the rendered document.

However, most wiki implementations are based on the traditional HTTP metaphor, by which the server returns the page to the requesting browser for rendering and does not participate in the rendering or receives further updates from the browser. Thus, such information can only be supplied by the client browser, which maintains the locations of each element under the current rendering, as well as the locations of the viewport during the session.

Unfortunately, making the browser collect and report all this information is difficult for a variety of technical and privacy reasons which we shall now describe.

2.2 Limitations of browser-instrumentation

Perhaps the most straightforward way to collect usage information from the client browser is to directly instrument it. Many modern browsers have plug-in mechanisms that can be used to listen and react to relevant internal events without changing the browser's source code. Such functionality could store the information locally, or transmit it to any external server for storage and analysis.

However, an obvious technical problem is the range of browsers types, versions, and platforms which must be supported. We would need to support and test our plug-in under all these settings and keep it up-to-date. Every additional plug-in also increases the risks of crashing the browser or of conflicts with other plug-ins. In addition, some users cannot install plug-ins due to organizational restrictions or limitations of their handheld devices. Even if these technical barriers could be overcome, we would be left with the problem of getting users to install the browser plug-in and update it when necessary. Even with incentives, the prospects for accomplishing this for all regular users are quite low, and are much lower for one-time visitors.

Furthermore, browser plug-ins have privacy implications and require the trust of the users since they have the theoretical capability of monitoring the use of every website, and not just our wiki. While it is straightforward to write a plug-in that reports only information related to a specific wiki, it is questionable whether users will actually trust it to not track and report the use of other sites, even inadvertently [3]. Also, many spyware detection and privacy assurance tools may be triggered by these reporting mechanisms, further alarming users. This is particularly likely if information is reported to a different server for storage.

2.3 Approach

Users may be more willing to deal with a monitoring scheme which does not require installation and which tangibly cannot track their activities outside the site it is supposed to be monitoring.

Our solution, depicted in Fig. 1, is based on a slight variation of the common AJAX technique. In AJAX, the web application embeds several JAVASCRIPT scripts in the page that it initially sends to the client. The client begins executing them and periodically queries the server for newer information that it then renders. In our solution, the scripts instruct the browser to install listeners to certain interactions, such as form data-entry and scrolling, and start a timer which, at regular intervals, collects information and reports it back for collection.

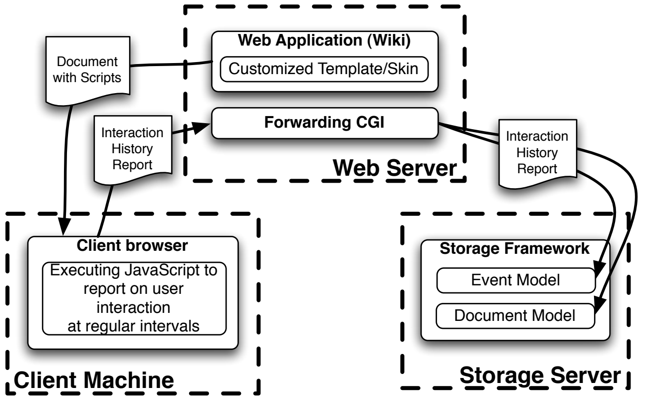

Figure 1. Proposed capture solution

However, since browsers do not allow scripts originating in one server to initiate connections to other servers because of security reasons, the information is sent to a CGI program on another path on the same server as the wiki application. This program can store the data locally for subsequent analysis, or route it immediately to a final storage and analysis service, such as the one described later in this paper.

This approach offers several significant advantages over browser instrumentation: First, It does not require users to install any plug-ins, since all the necessary code is embedded in the web page. Second, since the scripting languages are relatively standard, we can use the same code on the majority of browsers running on various platforms. Third, the scripts execute within the browser's sandbox, and are therefore less likely to crash it than plug-ins running natively on the operating system.

In addition, browser scripts are considered safe in that they cannot access data outside the browser, and do not persist when the user browses to a different location. They thus comply with the privacy expectations of users and do not trigger protection software. Of course, the collected data itself may have privacy reprecussions. However, since most wiki deployments are already capable of recording all page requests, the new data does not represent a fundamental difference in that regard. It may, however, enable a more accurate profiling of user activity that might need addressing in the site's privacy policy.

In the two years we have been using this approach to monitor wiki usage, others have independently used AJAX techniques to collect data from browsers. However, it appears that most of these efforts, such as the academic *UsaProxy* [1], and the commercial *TapeFailure*, *RobotPlay* and *ClickTale* tools, are all focused on capturing a low-level transcript of the session for usability purposes. Such recordings of mouse movements and keystrokes can be used to recreate an accurate playback of a session which is valuable in analyzing the site's visual design.

While our scripts can be augmented to capture the same information, low-level transcripts are generally too verbose and unstructured for our purposes. Our scripts are instead focused on gathering the information necessary to build the context, eliciting structural information and wiki-specific metadata from the page, and regularly emitting information about the viewport. They also report this information in a general form which is easier to correlate with traces of interactions from other sessions and other tools, as we shall describe in the next section.

2.4 Implementation details and status

Let us briefly discuss our current implementation of the approach described above. We have successfully implemented and tested the collection capability for the popular *MediaWiki*.[1] Adapting it to other wiki implementations should be straightforward to those familiar with their respective deployments and programming languages. Since many wikis defer much of their HTML generation functionality to a customizable skinning framework, scripts can be typically be embedded by modifying the skins rather than the core of the application. In *MediaWiki*, for example, this amounts to adding in every skin file a single `<?php include ?>` instruction, just prior to the skin's `</body>` element.

The included file, supplied with our deployment, will embed a script and references to several other script files in every generated page. Within the generated script, the file will explicitly embed the values of wiki-specific metadata that is difficult to obtain from the server after-the-fact, such as the page's version number, title, whether the page is in edit mode, etc. It will also embed some site-specific constants, including the address to which reports shall be sent, that the wiki owner will have to customize in advance.

When the browser receives the page from the server, it renders it, loads the additional script files (which are often cached), and begins executing the primary script. The script first generates a unique session identifier that is later used to identify related reports, and sends the first report. It then proceeds to repeatedly "sleep" and send an updated report upon waking up, until the session ends. All the reports are sent in XML form, to enable their use independently of our data storage framework, and to ensure compatibility with the traces we produce from other applications.

The first report includes the embedded metadata, and lists important elements and their relative offsets within the rendered document. If the page contains forms, as in the case that the user is editing, details are also supplied since later event logs may refer to the form controls. The contents of shorter documents can also be embedded in this report.

Since the document structure and metadata do not change during the session, subsequent reports contain only the current viewport in the form of offsets within the rendered document. By correlating this viewport with the offsets of the structure elements from the initial report, the collection framework can later ascertain what part of the document was visible at each point. When user activities are logged or form

[1] We have made an early unsecured version available online at `http://emoose.isri.cmu.edu/public/subpages/WikiLogging`

interactions take place, they are also reported as a series of events that occurred since the last report.

We note that the exact structure elements collected and reported by the scripts depend on the monitored application and the desired level of abstraction. Most wikis offer several levels of section headers, making sections natural candidates for representing the context. Using the location and level of each header in relation to other headers, it is possible to represent the document structure as a hierarchy of section spans. Our current implementation can identify relevant structure elements based on their HTML tags, identifiers, and classes. Surprisingly, while most web applications and many wiki implementations such as *SnipSnap* tag their section headers with classes to distinguish them from other HTML headers, that is not the case with certain popular implementations such as *MediaWiki* or *TWiki*. To allow section headers to be collected independently of other headers in the page, wiki owners may have to make minor modifications to their distribution's source code.

2.5 Limitations

The technique described above enables us to collect data from a variety of browsers on different platforms without requiring users to install any software, but the approach has several limitations that are worth noting. First, our approach relies heavily on the use of JAVASCRIPT as a standard language across all browsers. Unfortunately, even today the implementation of JAVASCRIPT and the associated document object model differs across browsers and platforms on some finer points. For example, there are differences in event listener registration mechanisms, and some browsers may not report viewports correctly.

A second limitation of our approach is that there is still some inherent "noise" in the information reported by the browser. The fact that a browser window is open on a specific wiki page does not always indicate that the user is actually reading its contents. In fact, the page could be obscured by another window, or be in an inactive tab of a tabbed browser, and continue to broadcast. It is not clear whether these problems can be adequately resolved without changes to the browser or installation of additional software, since JAVASCRIPT may not have the necessary facilities. We are currently exploring other techniques and heuristics to obtain this information.

A third limitation of our existing implementation is that it assumes that it is the only script that begins executing as soon as the page is loaded. Some modifications may be necessary for our scripts to work correctly without breaking pages that make heavy use of scripting and AJAX. Recent versions of the *UsaProxy* tool that are capable of instrumenting pages which already use AJAX demonstrate the feasibility of this goal.

Finally, we note that our current implementation does not address potential security problems, such as the sending of forged data to the target address. Collaborating between the script-generation and analysis logic is likely necessary to address these problems.

3. Representing usage data

Collecting usage data from a single wiki session is just the first step. Perhaps a greater challenge is to store it in a form that allows complex analyses and correlation with data from other sessions and from other sources. We try to meet this goal with our *Episodic Memory Of Open Source Efforts* (*eMoose*) framework[2]. This extensible framework is designed to represent "memories" of activities that involve multiple tools. In addition to a common representation of activity traces, *eMoose* maintains structural and temporal information about the documents or artifacts involved in the activity. It can also manage additional metadata and multimedia that forms part of these so-called memories, such as multimedia recordings. It is designed to allow, given comprehensive enough data, the reproduction of a playback of all the activities involved in a development task, while offering sufficient structure to allow effective querying and data aggregation.

3.1 Framework Architecture

The framework was designed to facilitate the creation of new data collectors, retrieval mechanisms, and client applications, while supporting multiple platforms and database formats, even at the cost of lower runtime performance.

At its core lies the *persistence kernel*, an interchangeable layer of software responsible for maintaining the collected data using different technologies and representations, such as various database managers, file systems, or even the memory of clustered machines. The kernel can only be accessed via a set of JAVA interfaces which represent entities in our abstract data model, and which support certain types of queries. This approach allows us to support a variety of platforms and make various space and performance tradeoffs to fit different organizations, in a manner transparent to clients.

Components that supply data, such as the servlet that receives the reports from the wiki or the daemon that receives traces from the IDE, create representations of this data using these interfaces and then instruct the kernel to persist them.

Components that consume data, such as client applications and report generators, can directly query the persistence kernel based on these interfaces. However, our native data model may be too comprehensive and complex for certain applications, such as the generation of simple wiki usage reports that are not concerned with activity in other application. To this end, components called *model presenters* serve as a buffer between the kernel and the applications, and can offer a simplified domain-specific view of the data. For example, one can foresee a custom model presenter which offers a single `WikiSession` interface that provides the time,

[2] This framework was developed as part of the author's ongoing dissertation work which investigates the role of memory and context in software development.

duration, and target page of each session, and the relative time spent in each section of the page. When this presenter is used, information requests are translated behind the scenes to operations on the standard interfaces.

3.2 Data model

To illustrate the capabilities of the native data model, let us briefly survey its interfaces, which can conceptually be divided into three parts: updates, events, and documents. The update interfaces are primarily concerned with the bookkeeping involved in incremental data collection, since an application's monitoring data may arrive out of order or from multiple sources.

Event objects are the discrete elements of an interaction trace. In the case of wiki data, each periodic report results in an event that represents the fact that at that point in time, the page was open and the viewport was at a certain location. Event objects are associated with sessions and can relate to other events to form hierarchies of aggregation, allowing us to define higher levels of abstraction. Events can have named parameters to store additional metadata; these may include specific references to documents. For example, every event from the wiki will refer to a *document snapshot* to which the viewport applies.

The uniqueness of *eMoose* over the simple event traces captured by other tools lies in its complex document model, which offers context to the activities represented by these traces. A *document* is an abstraction of the identity of a resource which can change over time, such as a source file, a bug report, or a wiki article. A *document snapshot* is an abstraction that represents the state of a document at a given time when viewed in a certain manner. For example, a specific wiki article at a certain time, as obtained via a specific URL and rendered on a specific browser on a specific machine is a snapshot. The state of a document on a knowledge worker's spreadsheet, or of a source file on a software developer's machine at a certain time is also considered a snapshot. Snapshots are typically accompanied by structural information or actual document contents.

Though snapshots are a continuous abstraction, *eMoose* collects only sporadic and discrete instances. For example, the state of a wiki page is captured when it is first rendered since it does not change afterwards, and the state of a source code file is captured when a significant change has occurred. In many cases, missing snapshots can be extrapolated by combining concrete snapshots and events.

While snapshots are stored independently from events, they can be correlated with them by session and timestamp. One could query, for example, for all the sessions in which a certain section of a certain wiki page was read, and then query for the visible source files in the IDE at these times.

4. Conclusions

In this paper we motivated the need for a capability of tracking wiki usage from the client side, and highlighted its importance for research on information use in knowledge work, and in software development in particular. We also described a technical solution for eliciting usage data without requiring users to install or instrument their browsers, and a framework for storing and querying this data along with similar data from other applications.

At the time of writing, we have not been able to recruit collaboration in validating our solution on wikis in the field. However, we have successfully used the same technical approach to instrument a non-wiki web application, the *Bugzilla* tool which is used for bug and issue tracking in many open source projects. In addition, we have developed an extensive monitoring functionality for the popular *Eclipse* IDE which reports low level events, and an independent automated functionality which identifies higher-level events in these streams. The traces from all these applications were successfully stored and retrieved together from a single *eMoose* repository.

We are currently developing a tool for providing software developers with a continuous view of a "recollection" of recent events, including the use of wiki pages. We are also designing an experiment to study the potential benefits of this information to reducing disorientation during development, and to help in hand-off scenarios, where another developer must maintain an existing program.

Acknowledgments

This work was supported by NSF grant IIS-0414698, and the CMU Software Industry Center.

The author would also like to thank Arunkumar Jayaraman, Aarthi Prasad, Lakshmi Rajamohan, and Sivaparamesh Ravindran for their contributions to the development of the *eMoose* software.

References

[1] R. Atterer et al. Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction. In *WWW '06*, pages 203–212.

[2] G. Beenen et al. Using social psychology to motivate contributions to online communities. In CSCW'04, pages 212–221.

[3] K. Hawkeye. Mission impossible? capturing rich yet natural user behaviour on the web. In *Workshop on Logging Traces of Web Activity at WWW'06*.

[4] A. J. Ko et al. Information needs in collocated software development teams. In *ICSE '07*, pages 344–353.

[5] T. D. LaToza et al. Program comprehension as fact finding. In FSE '07, to appear.

[6] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06*, pages 492–501.

[7] S. Reinhold. Wikitrails: augmenting wiki structure for collaborative, interdisciplinary learning. In *WikiSym '06*, pages 47–58.