Referencing within evolving hypertext

Victor Grishchenko, Janus A. Pouwelse, and Henk Sips

Delft University of Technology
Mekelweg 4, 2628CD
Delft, The Netherlands
victor.grishchenko@gmail.com

Abstract. The classic hypertext model omits the process of text growth, evolution and synthesis. With hypertext creation becoming increasingly collaborative and change timescales becoming shorter, explicitly addressing text evolution is the key to the next stage of hypertext development. Uniform Resource Identifier (URI) is a proven general concept that enabled the Web. In application to versioned *deep* hypertext, expressive power of a classical hyperlink becomes insufficient.
Based on the Causal Trees model, we introduce a minimalistic but powerful query language of *specifiers* that provides us great flexibility of referencing within a changing hypertext. Specifiers capture the state of the text, point at changes, expose authorship or blend branches. Being a part of an URI, a specifier puts advanced distributed revision control techniques within reach of a regular web user.

1 Introduction

In the WWW/HTML model and, generally, in "chunked hypertext" systems the main addressable unit is a "page" which might optionally also have addressable "anchors" inside it. That is generally sufficient as long as we deal with static texts, albeit the requirement that a page author must pre-provision anchors is limiting. However, if we follow the general vision of a text as an evolving entity (the "wiki model"), then the expressive power of a classical hyperlink is insufficient. Since the addressed text is continuously changing, anchors might disappear, and the content that is actually addressed by the link might be reedited or its surroundings may change. Similarly, there is no standard way of pointing at particular statements and passages in the text. For collaboratively created texts, such a possibility is desirable. Also, there is no semantics in place to address co-existing versions of a text (named "branches" in the version control parlance). Those might be drafts, reeditions, alternative versions. Thus, our mission is to explore possible approaches and conventions of referencing particular parts of text, its particular versions, or both. We want to measure, mark and cut text in breadth and depth!

This paper is structured as follows. First, we consider relevant existing models and their limitations in Section 2. Section 3 briefly describes the Causal Trees (*ct*) model of text versioning and the basic primitives available for text/ version

addressing. In Section 4, based on the URI specification, we define the syntax of specifiers.In Section 5 we consider practical applications for the proposed conventions, explained as simple Alice-Bob scenarios. The Section 6 concludes.

2 Related and previous work

The early hypertext system Xanadu employed Dewey-inspired change-resistant addresses named tumblers, e.g. `1.2368.792.6.0.6974.383.1988.352.0.75.2.0.1.9287` (an example from [21] addressing a particular point in a particular version of a document). That scheme was not reused by any later system.

Today, most wikis, including Wikipedia, have a history view capable of retrieving and comparing different versions of a text. However, the URL syntax is implementation-dependent. The authors are unaware of any wiki that allows for branching/ multiversioning of texts; a document's history is always seen as a linear sequence of numbered revisions. Distributed revision control systems[1] implement an extensive toolset for identifying/processing parallel revisions of texts (typically, source code). Most of those systems model a mutation history as a directed acyclic graph of revisions. The inner contents of a file are considered a single data piece; no fine-grained addressing is possible. Revisions are typically identified by cryptographic hashes of the content and metadata. A number of wikis[2] use distributed version control systems as their back-end, but they don't pass on the branching functionality to the front-end.

The possibility of addressing precise parts of a text is a well-known general problem. Texts that need repeated reading, referencing or modification typically have some fragment addressing scheme as well. Examples are Biblical (e.g. `1 Kings 11:41`), Qur'an (`2:2`), legal (`U.S.Const.am.8.`) or source code (`kernel/panic.c:57`) references. Paper books are referenced using page numbers, but those might change from edition to edition. These days, various e-book devices made the notion of "a page" completely ephemeral. In application to computer hypertext, three classic examples of addressing schemes are Purple numbers [16], XPointer [7] and the classic `patch` [6] format. They rely on three basic techniques: offsets, anchors and/or context. The first and the simplest addressing technique is to use word/symbol offsets within a file. That works well for static files. For example, web search engines employ inverted indexes that list all the document-offset pairs where a particular word was found. But, in a changing text, new edits invalidate offsets. Hence, every next version has to be processed as a separate text. The second technique is planting anchors within the text. However, pre-provisioning anchors for any future use by any third party is not practical. The third technique is to address a point in the text by mentioning its *context*, i.e. snippets of surrounding text. The approach is robust, but heavyweight, dependent on heuristics and also vulnerable to text mutations. Combining those techniques may increase robustness, e.g. the UNIX

[1] For example, Bazaar http://bazaar.canonical.com/, Git http://git-scm.com/, Mercurial http://mercurial.selenic.com/

[2] For example, Gitit wiki http://gitit.net or git-wiki http://github.com/sr/git-wiki

`diff/patch` format employs approximate offsets *and* context snippets. But, that might increase fragility as well; e.g. an XPointer relying on an anchor *and* an offset becomes vulnerable to changes in both. Purple numbers address paragraphs using either offsets or anchors. Neither method is perfect.

Several well-known technologies, such as WebDAV [13], BigTable [10] or Memento [23], represent history of an evolving Web page as a sequence (or a graph) of revisions identified by either timestamps or arbitrary labels. In that model, every version stays a separate monolithic piece. There are some efforts to apply versioning to adaptive hypertext [17].

The Operational Transformation theory (OT, [11]) generalized offset-based addressing scheme for changing texts with the purpose of real-time revision control in distributed systems. Currently OT is employed by Google Docs, Google Wave and other projects[3]. Among the shortcomings of the OT theory is its high complexity and long-standing correctness issues [19, 15]. Systems that are known to work had to adopt compromises on the original problem statement [11], either by relying on a central coordinating entity [3] or by requiring that edits are always merged in the same exact order [22]. Still, the main problem is bigger: OT does not address revision-control tasks that lie outside the frame of real-time collaborative editing, narrowly defined. Those are: branching, merging, propagation of changes, "blame maps", diffs and others.

The Causal Trees (*ct*) model [14] was introduced to resolve the problems evident in OT. Instead of relying on offset-based addressing, which is volatile once we consider a changing text, *ct* assigns unique identifiers to all symbols of the text. Thus, it trivially resolves the correctness/complexity problems and also introduces new possibilities. For example, it allows fine-grained fragment addressing that survives edits. Being defined along the lines of the Lamport-Fidge [18, 12] time/event model, it allows for reliable identification of any versions, even in a text that has multiple concurrently changing editions (branches). Effectively, *ct* implements the functionality of deep hypertext, as described in [1]. The *ct* model is the starting point of this work. Recently, the *ct* model was implemented as a JavaScript library *ctre.js*[4].

Consider a document which has several evolving branches. Suppose it is a Wikipedia-style wiki of course materials which is supported by several universities in parallel. On the one hand, we want to maintain the upside of collaboration which is well illustrated by the success of Wikipedia. On the other hand, we want to avoid edit warring [2] and the extreme volatility of content typical of Wikipedia. Thus, we suppose that such a wiki is supported by a network of collaborators exchanging, negotiating and filtering edits in the way the Linux kernel is developed (the "git model").

In such an environment, a single document may be seen in hundreds of ways, depending on which editions we are looking at, and when. Similarly, if we want to address some particular parts of a document, there are numerous possibilities.

[3] Google Docs http://docs.google.com, Google Wave http://wave.google.com, Gobby http://gobby.0x539.de, Etherpad http://etherpad.org

[4] Project page at GitHub: http://github.com/gritzko/ctre

Our intention is to extend the semantics of URIs [8] to deal with that complexity and to make it manageable and clear to a regular user. We assume that the address field of the browser is the user's primary means of navigation. Finally, the ability to identify and address (and instantly access) arbitrary resources is the cornerstone of the Web. We extend that ability in space and time.

3 The ct model

The *ct* model augments the very fabric of text to reflect its evolution. Drawing some lessons from the history of Operational Transformation theory, *ct* does not use offset-based addressing and does not try to find one true frame of reference. Instead, *ct* closely follows the lines of the Lamport-Fidge [18, 12] relativistic model of events and time in a distributed system. In essence, *ct* is a Minkowski *spacetime* [20] model for versioned texts, unifying time (versions) and space (text) as different projections of the same phenomenon. This section will briefly explain the basics of the model and its building blocks.

A. Symbols have own identity. Attempts to identify symbols by their offsets in a versioned text have produced unsatisfactory results. As a text constantly changes, so do offsets. Thus, it becomes nearly impossible to reliably *point at* a given character. Instead, *ct* starts by assigning unique identifiers to every symbol in a document. Securing globally consistent serial identifiers is impossible in a truly decentralized system, thus we resort to the Lamport-Fidge approach. For a given document, all its symbols originating from the same author are sequentially numbered. Thus, they constitute a vector of contributions (called a *yarn*) of that particular author to that particular document. Still, we do not try to impose any global numbering. Instead, we identify a symbol by its (yarn_uri, symbol_number) pair, i.e. `(alice.org/page,398)`. Given that id, we may always retrieve the symbol, and a symbol may be reliably pointed at, independently of any changes in the document. A symbol with an identity is called an *atom*.

B. Text and operations are the same. There is no separation of "a text" and text-modifying "operations". A text consists of *atoms* and any operation is a set of atoms as well. An "atom" is a symbol plus its metadata. Even deletions are implemented as special "backspace" meta-symbols. An atom's metadata consists of its own identifier and an identifier of the *causing* atom. The causality relation weaves atoms together to form a text. Very much in the spirit of the Markov chain [9] model, a symbol is said to be *caused* by its preceding symbol at the time of insertion. Such a simple relation leads to provable correctness and convergence even in a distributed system with no central entities [14]. All replicas of a text eventually converge to the same state and no edits are misapplied.

C. All frames of reference are equal. A frame of reference corresponds to a single "local" author and his version of the text. There is no special "central", "reference" or "true" version. When accessing a text, we access not the text per

se, but its version by a particular editor (a yarn). Other yarns are retrieved by recursively following causal dependencies. Then, yarns are woven together to produce a version of the text [14]. A transitive closure of causal dependencies is one of the key concepts of *ct*. For example, it defines the way branches are represented in *ct*. A branch is a set of yarns that has dependencies on the "trunk" yarns of the text, but there are no dependencies in the reverse direction (trunk to branch). Once such reverse dependencies are created, the branch effectively merges into the trunk, as it becomes a part of the trunk's closure.

3.1 Unicode serialization

In practice, *ct* is implemented with regular expressions[5]. That is not only mathematically well-defined, but also practically useful, as it allows to run *ct* in a Web browser with native speed. To make atoms regex-processable, their ids are encoded with two Unicode symbols, one for the author/yarn and another for the symbol's serial number. For example, an atom id `(alice.org/page,398)` becomes "AΘ", where "Θ" is the Greek capital letter Theta corresponding to Unicode code point 398. We also assume a mapping between symbols and URIs, where "A" corresponds to `alice.org/page`.

While two-symbol ids might seem insufficient, they allow for up to 4 billion symbols per document if using a baseline regex implementation supporting only 16-bit Unicode BMP characters [5]. Once an author exceeds the 2^{16} symbol limit, (s)he might be allocated another yarn id. The case of 2^{16} authors per document is considered highly unlikely, and even if that happens, there is always an option to use two characters for a yarn id (hence three characters per atom id total). Still, we believe that the two-symbol scheme provides sufficient numbering capacity for most of the texts.

Serializing atoms as tuples of Unicode symbols allows to pack all data structures into strings and to process them with regular expressions. That resolves an important practical bottleneck. Performing sophisticated revision control operations in a Web browser, in real time, becomes possible.

3.2 Specifying ranges and versions

In full accordance with the *spacetime* concept, *ct* denotes intervals in time (versions) and intervals in space (text ranges) in a very similar way, namely by mentioning their bounding atoms. Regarding text ranges, we may rely on the linear order of symbols in a text, and simply denote a text range by mentioning its end-points. Thus, $[A4; B8)$ stands for an interval starting at a symbol number 4 by author A and lasting till, but not including, the symbol number 8 by B.[6] This interval specification is immune to any further text changes, including deletion of the bounding atoms.

[5] The PCRE (Perl Compatible Regular Expressions) dialect, as used in JavaScript

[6] Parentheses () stand for excluded endpoints, square brackets [] for included.

Denoting versions in a distributed system may be trickier. In the simplest case, a version history is linear (e.g. there is only one author). Then, a revision may be denoted just by mentioning its most recently introduced symbol, e.g. $B8$. Thus, that one and all the "older" symbols $B1$–$B7$ constitute a version. But, in the general case of a distributed system, all participants are free to introduce new changes, and those changes propagate with finite speed. At a given moment in time, each participant sees a version of a text, based on the edits it is aware of. Thus, version history is not sequential. The only appropriate model is a directed acyclic graph. In such a case, a version might have multiple "most recent" symbols, e.g. $A4$ and $B8$. The number of such symbols cannot exceed the number of authors or, more precisely, yarns. Essentially, a set of those "most recent" symbols is a logical vector timestamp [18]. Under the hood, *ct* represents vector timestamps as *wefts*, which are strings of even length consisting of two-symbol atom identifiers, like `A4B8`.

Note that range/version specifications may have any of their bounding symbols either excluded or included. In general, that time-space unification will help us a lot with our specifier syntax (see Sec. 4.2).

4 URL conventions

An atom identifier is just a pair of Unicode symbols, one for the author/yarn code and another for the atom serial number within the yarn. A version or a text range is thus denoted by several Unicode symbol pairs. We want to use them in URIs as basic building blocks of our version/fragment specifiers. But, the URI syntax [8] only allows for a restricted subset of 7-bit ASCII symbols. So, we have to define a serialization of atom identifiers. Our end goal is to develop a syntax convention that allows to denote text versions and ranges by URIs that are easily communicated using email, instant messaging, Twitter, napkins [8] and even spoken speech.

4.1 Encoding

The default option for using Unicode in URIs is the percent-encoding [8]. But that would consume six characters per every non-ASCII symbol (hence, 12 per atom id). Instead, we encode atom identifiers using base64 encoding, namely its variety that employs alphanumerics, tilde and underscore to express $64 = 2^6$ values.[7] We only consider Unicode characters of the Basic Multilingual Plane, which gives us 2^{16} possible values for either author or symbol code, and correspondingly 2^{32} possible values of an atom id. We resort to separate encoding of author and symbol codes by up to three base64 characters each ($2^{6\times3} > 2^{16}$). Thus, an encoded atom id may take *up to* six base64 characters.

We expect most texts to be short, created by a few authors, so most author/symbol codes will have low values. As one URI may include many atom

[7] `0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz~_`

ids, it is highly beneficial to use a variable-length encoding to shorten serialized atom ids, when possible. In the worst case, we still use 6 symbols per atom id, but in case, for example, we see a three-symbol id, we know that the first symbol stands for the author and the other two for the symbol code. Assuming that atom ids are guaranteed to be bounded by delimiter symbols, we may agree to use `2=1+1`, `3=1+2`, `4=1+3`, `5=2+3` and `6=3+3` conventions. For example, suppose an atom id is encoded with five base64 characters: `0e5ZC`. Then, according to the `5=2+3` convention, the first two stand for the author code (up to $2^{2\times6}$ values) and the next three stand for the symbol's number (up to $2^{3\times6}$ values). Numerically, `0e` $= 0 \times 64^1 + 40 \times 64^0 + 48 = 88$ is the code for the author and `5ZC` $= 5 \times 64^2 + 35 \times 64^1 + 12 \times 64^0 + 48 = 22780$ is the serial number of the symbol among those contributed by that particular author to that particular page. (Unicode code point 88 corresponds to a Latin capital letter `X`. Code point 22780 corresponds to a hieroglyph 壼).

To express the aforementioned semantics of included/excluded bounds (see Sec. 3.2) and to guarantee reliable delimiters, we prepend every atom identifier with either `+` or `-`, denoting included or excluded bounding symbols respectively, e.g. `+0e5ZC`. The default value is `+`. We also allow atoms to be marked with mnemonic labels. Then, instead of a base64 representation of an atom id, we will use a single-quoted alphanumeric label, e.g. `+'SOME_VERSION'`.

While every author/yarn is encoded with an arbitrary Unicode char, it is convenient to make base64 representations somewhat meaningful semantically. Thus, instead of encoding "Alice" as `0e` (i.e. `88` or Unicode `X`), we will try to use such options as `Alc` (Unicode 갋), `Al` (Unicode ˣ) or simply `A` (Unicode `:`).

4.2 Text state/presentation specifier

Under the hood, the *ct* model has lots of version control related features and functions. Indeed, there are hundreds of ways to display an evolving text, and many of them are useful. The main bottleneck is the user's ability to perceive that data and to access that functionality. So, the core of our mission is to put that toolset within reach of an end user. Practically, we develop a small query language based on URI-embeddable expressions that will allow us to access the most of *ct*'s capacity right from the browser's address bar.

A *specifier* is a complete URI-embedded expression describing the desired state of the text and nuances of its decoration. A specifier contains a sequence of *parameters*. Each parameter affects a single aspect of text state or text presentation. A "state" parameter changes the actual text body delivered to the user, while a "presentation" parameter only adjusts its decoration (i.e. color, highlighting, strike-through, other marks). In this section we describe seven types of parameters: three state, three presentation and one mixed type.

The space/time unification helps us a lot. It lets all parameters follow the same syntax convention with minor variations. Every parameter starts with a special separator symbol (typically a sub- or general delimiter in terms of [8]). The separator defines the type of the parameter. The separator is followed by a sequence of zero or more atom identifiers and/or quoted labels.

Version is the first and the most basic state parameter. It defines the version of the text that is actually shown to the user. A version parameter employs exclamation mark `!` as a separator, normally followed by atom identifiers. Suppose, Alice wrote "Hallo wrld" and Bob corrected that to "Hello world". Thus Alice contributed 10 atoms (say, `Al01-Al0A`) and Bob contributed three (including one backspace, e.g. `Bo01-Bo03`). Then, the resulting version is `!Al0A+Bo03`.

Range is a state parameter specifying a fragment of a page that has to be delivered to the user. Range separator is `:` (a colon). In the example above, a range specifier `:Al01-Al06` initially points at "Hallo". Once Bob fixes the typo, the value of the same range changes to "Hello".

Branch is a parameter that allows to work with parallel versions of the same text. The separator is `=` (equal sign). A branch might be specified either with a label or with a yarn id, i.e. `='Branch'` may be interchangeable with `=Br`. The default "trunk" branch is addressed as `=`. The *ct* model allows to deal with branches in completely novel ways. In particular, it allows to merge (blend) branches in real time. Our syntax should let users access that functionality. In case multiple branches are specified, their contents are merged (blended), but all new edits go to the first mentioned branch. So, a specifier `='Draft'=` merges the trunk with the Draft branch, but all new edits go to the latter.

Fragment is a presentation parameter analogous to the range parameter. It specifies an area of interest within the delivered text. We re-use the standard URI fragment separator `#` (number sign). Important detail: the fragment part of URI is not reported to a HTTP server by a HTTP client (i.e. the browser). Hence, all corresponding actions are performed locally in the web browser (i.e. page scroll or range highlighting). In our example, `#Bo02-Al05` would show "Hello world" with "ell" selected or highlighted.

Baseline version is a presentation parameter pointing out which version is considered "the previous version". Thus, all changes that happened after that "previous" reference version should be highlighted. That is most useful when a user wants to see the changes that happened since his/her last visit, or otherwise compares two versions. This parameter employs `$` as a separator. Thus, to see a difference between two versions, Alice may use a specifier like `$Al06!Al0A`. That will deliver "Hallo wrld", with "wrld" highlighted.

Author parameter suggests to somehow mark/unmark contributions of certain authors. The separator is `@` (at sign). For example, `$Al08@-'Alice'` will unmark contributions made by Alice thus only leaving "e" and "o" highlighted, as those are contributed by Bob: "Hello world". Here we deviate from the general scheme of using full atom identifiers after a separator, as we only need to identify an author/yarn (the same as with branches). We may rewrite the same specifier as `$Al08@-Al`.

Change status is a mixed state/presentation parameter with a syntax somewhat deviating from the common pattern. It employs an asterisk `*` as a separator. Its mission is to filter/recover symbols based on their insertion/deletion status. For example, `$Al0A*+AlBo` shows all symbols inserted by Alice and removed by Bob since version `!Al0A`. Thus, the resulting text is "He~~a~~llo world", with "a" struck out, "e" and "o" highlighted.

Effectively, we created a small query language that controls state and presentation of a versioned text, points at ranges and versions, locates changes, navigates branches. As with any language, the expressive power comes from combining the primitives. We may easily imagine sophisticated but still comprehensible constructions, like:

```
http://server.dom/Proposal='Draft'$Alzu!Bo4Vk@-Bo#Alb8-AlyK
```

That means: "on a page named Proposal, within a branch named Draft, using version `Bo4Vk`, please highlight changes made since version `Alzu`, except for the changes made by Bob, and please select the range `Alb8`–`AlyK`". We do not expect every user to master this language. Composition of queries may be done by the GUI. Still, we see that this formal and compressed way of expressing versioning-related page state/presentation opens promising possibilities. One interesting example is the ability of specifiers to fully describe the current state of the edited page, including the current selection. If all changes of the state are reflected in the fragment part of the URI (rewriting fragment does not cause the page to reload), then the entire page state may be copied and sent by e-mail or IM to another person. An evolving text is almost like a river, in the sense that you cannot step into the same river twice. With specifiers, remote collaborators will have that possibility to be almost literally *on the same page*.

5 Scenarios

In this section we consider a hypothetic scenario of Happytown State University participating in a project that might be briefly described as a cross between OpenCourseWare and Wikipedia, collaboratively developed the git way. Collaborators from peer universities contribute academic information, including course materials, lecture notes and general articles. Users experience the system as a real-time wiki running in a browser. Each university hosts its own wiki.

That wiki also supports branches and distributed revision control to allow for parallel coexistence of drafts and working versions, on par with polished "canonical" public versions. Distributed revision control also allows to *federate* wikis of different universities. Users may pull changes from peer wikis in automated or manual fashion. Eventually, constant exchange of changes makes different wikis converge. Still, some pages may differ, because they are not merged yet or because of a conflict, e.g. in case Prof. Montague is unable to find common ground with Prof. Capuleti.

Voluntary import of changes allows to avoid Wikipedia-style edit warring and "the most persistent person wins" problem. It also acts as a soft variety of peer review, improving prestige of authors whose edits are widely accepted. While changes propagate from site to site, all authorship information is preserved intact. Academic prestige provides healthy incentives for participation, while direct spamming and self-promotion are countered by social filtering. Effectively, we consider a hypothetical bottom-up open-source academic publishing system.

5.1 Lecture and scribes

While assistant professor Alice delivers a lecture, appointed scribes transcribe it collaboratively in real-time by filling the lecture skeleton previously created by Alice. As the course is available on the web site both to peer universities and general public in real-time, scribes use a separate branch for their work: `http://ocw.happytown.edu/Lecture=Scribes`. Once scribes enter the URL, the branch is automatically created. They transcribe the lecture quite hastily. After the lecture, PhD student Bob polishes the text and merges it back into the university's trunk version. Thus, the public version is now updated and available to external audiences. Later, PhD student Fred of Fartown University decides to cite a passage from the lecture. He selects the passage and uses the link from his browser's address bar: `http://ocw.fartown.edu/Lecture#Bk-B7~`
Note that Fred uses the Fartown wiki which pulled content from Happytown.

5.2 Private remarks

Professor Carol oversees the lecture to see how well Alice is doing. Carol wants to see what scribes are recording to avoid getting out of sync with their version. Still, Carol wants to keep her remarks private. Thus, Carol blends `Scribes` with her own private branch by entering URL: `http://ocw.happytown.edu/Lecture=Notes=Scribes`. Now, the edits made by the scribes and her own changes are visible to Carol as a single merged text, updating in real time. Authorship is highlighted, one color per a yarn. Scribes cannot see the remarks made by the professor. Later, Carol will discuss the `Notes` with Alice and they will work on improving the text. A polished version of the branch will have to be merged back into the trunk. But, the edit history of the branch has lots of offhand remarks and back-and-forth editing which shouldn't go into the public history of the document. Thus, Carol *rebases* [4] the changes into the trunk, i.e. includes them by value, not by reference, leaving the edit history behind.

5.3 Back from vacation

PhD student Bob gets back from a conference and a vacation and logs into the system. He finds out that the project has advanced a lot since he departed. He loads the project status page. The changes made since his last visit are highlighted, contributions of different authors shown in different colors. At some

point Bob decides to discuss the changes with post-doc Dave, who has authored some key new pieces of the text. Bob wants Dave to see exactly the same "blame map" highlighting as well. He picks the full URL of the current page state to paste it into an instant messaging client `http://ocw.happytown.edu/Lecture$Bo2j8` Then, he understands he should ask a focused question, so he selects a passage he is mostly concerned about. The URL changes to `http://ocw.happytown.edu/Lecture$Bo2j8#Bo64Q-D77j` now denoting the selection. Bob pastes the link into IM asking Dave for a clarification. This way he minimizes interruptions of context that would otherwise result from going and asking about "that change".

6 Conclusion

In this work, we bridged the *ct* model [14] of deep hypertext and the generic URI scheme [8]. We have shown that a simple and laconic convention may provide very fine-grained addressing for particular versions and/or segments of a changing text. Although the convention is *ct*-specific, it is rooted in the Lamport-Fidge time/event model and thus more predetermined than arbitrary.

We described several novel end-user scenarios for deep hypertext applications. Currently, distributed revision control is an expert-only domain. We have shown that such a functionality might be served to a regular academic user as well.

7 Acknowledgements

This work was partially supported by the EC FP7 project P2P-Next, grant #216217. Authors are grateful to David Hales for valuable feedback.

References

1. Deep hypertext: The Xanadu model. `http://www.xanadu.com/xuTheModel/`.
2. Edit warring on Wikipedia. `http://en.wikipedia.org/wiki/Wikipedia:Edit_warring`.
3. Google Wave protocol. `http://waveprotocol.org`.
4. Rebasing at "The Git Community Book". `http://book.git-scm.com/4_rebasing.html`.
5. The Unicode standard. version 6.0 - core specification. `http://www.unicode.org/versions/Unicode6.0.0/`.
6. Unix manual page for the patch utility. `man patch`.
7. XML Pointer Language (XPointer) Version 1.0. `http://www.w3.org/TR/WD-xptr`.
8. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax, 2005.
9. Markov A.A. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete, 15:135—156, 1906.
10. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

11. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. SIGMOD Rec., 18:399–407, 1989.
12. Colin Fidge. Logical time in distributed computing systems. Computer, 24(8):28–33, 1991.
13. Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP extensions for distributed authoring – WEBDAV. RFC 2518.
14. Victor Grishchenko. Deep hypertext with embedded revision control implemented in regular expressions. In Proceedings of the 6th International Symposium on Wikis and Open Collaboration, WikiSym '10, pages 3:1–3:10, New York, NY, USA, 2010. ACM.
15. Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In ECSCW'03: Proceedings of the Eighth European Conference on Computer Supported Cooperative Work, pages 277–293, Norwell, MA, USA, 2003. Kluwer Academic Publishers.
16. E. E. Kim. An introduction to Purple. `http://eekim.com/software/purple/purple.html`.
17. E. Knutov, P. De Bra, and M. Pechenizkiy. Versioning in Adaptive Hypermedia. In Proceedings of the 1st DAH'2009 Workshop on Dynamic and Adaptive Hypertext: Generic Frameworks, Approaches and Techniques, pages 61–71, 2009.
18. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7):558–565, 1978.
19. Du Li and Rui Li. An admissibility-based operational transformation framework for collaborative editing systems. Comput. Supported Coop. Work, 19:1–43, February 2010.
20. Hermann Minkowski. Raum und Zeit. B. G. Teubner, Leipzig, 1909.
21. Theodor Holm Nelson. Literary Machines. Mindful Press, 1982.
22. Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Trans. Comput.-Hum. Interact., 5(1):63–108, 1998.
23. H. VandeSompel, M. Nelson, and R. Sanderson. HTTP framework for time-based access to resource states – Memento. `draft-vandesompel-memento-01`.