Differential Synchronization

Neil Fraser
Google
1600 Amphitheatre Parkway
Mountain View, CA, 94043
fraser@google.com

ABSTRACT

This paper describes the Differential Synchronization (DS) method for keeping documents synchronized. The key feature of DS is that it is simple and well suited for use in both novel and existing state-based applications without requiring application redesign. DS uses deltas to make efficient use of bandwidth, and is fault-tolerant, allowing copies to converge in spite of occasional errors. We consider practical implementation of DS and describe some techniques to improve its performance in a browser environment.

Categories and Subject Descriptors

H.5.3 [**Group and Organization Interfaces**]: Collaborative computing
I.7.1 [**Document and Text Processing**]: Version control

General Terms: Algorithms, Performance, Reliability.

Keywords: Synchronization, Collaboration.

1. INTRODUCTION

The increased availability of always-on Internet connections has increased the demand for applications which allow multiple users to collaborate with each other in real-time. Many such applications exist, including Google Docs, SubEthaEdit and Mozilla Bespin. At the heart of each application is a choice of synchronization algorithm. In our experience this choice is usually made early in the development cycle, is very difficult to change later, and has a major impact on all the operating characteristics of the application.

In this paper we present Differential Synchronization (DS), which is a minimalistic synchronization mechanism, whose design goal is to have minimal impact on application design. This goal also makes DS suited for use in existing applications.

DS is a state-based optimistic synchronization algorithm.[13] The synchronization topology is a tree, with changes converging on inner nodes, and thus no elaborate versioning model is needed to capture causality. Concurrent changes are reconciled by patching the changes from one peer into the copy on another. Changes are detected by differencing the current state against the previously established state, yielding a diff. Updates are propagated between peers as that diff.

DocEng'09, September 16–18, 2009, Munich, Germany.

This algorithm's main attributes are:

- Symmetrical with (nearly) identical code running on both client and server.
- It is state-based and does thus not require that applications maintain a history of edits.
- Asynchronous, which eliminates blocking user input while waiting for a response over the network.
- Forgiving of unreliable and high-latency networks.
- Convergent, errors do not cause different copies to diverge.
- Suitable for any content for which semantic diff and patch algorithms exist.
- Highly scalable.

As of this writing, the major users of DS are the set of code editors which use or are compatible with MobWrite[4], such as Eclipse, Bespin[1] and Gedit[2]. Using DS as a standard synchronization system between them allows users of one editor to collaborate with users of any other editor. Typical use cases include pair programming between distributed sites, the ability to invite a remote expert to debug some code in an active session, and enabling employers to watch and interact with code being written by a candidate during a telephone job interview.

One unexpected use for DS involves online applications desiring autosave functionality. In such applications it is not uncommon for a user to inadvertently end up collaborating with himself:

1. User makes changes using a desktop. System autosaves.
2. User opens application with a laptop. System restores from autosave.
3. User makes changes from the laptop. System autosaves.
4. User switches back to the desktop which had been left open. System autosaves, destroying work done on the laptop.

With DS keeping all open versions constantly in sync, the user's keystrokes are mirrored in close to real-time across all his terminals. Thus the user does not need to be aware of his workflow and can submit the content from any terminal while being assured that all edits are present.

[1] https://bespin.mozilla.com/

[2] http://groups.google.com/group/patchworkeditor

2. ALTERNATIVE STRATEGIES

Three common approaches to synchronization are the pessimistic approach, edit-based and three-way merges. These methods are conceptually simple, but all have drawbacks.

2.1 Pessimistic

The pessimistic approach is the simplest. In its most basic form, a shared document may only be edited by one user at a time. A familiar example is Microsoft Word's behaviour when opening a document on a networked drive.[9] The first user to open the document has global write access, while all others have read-only access. This does not allow for real-time collaboration by multiple users.

A refinement would be to dynamically lock and release subsections of the document. However this still prevents close collaboration. Subsection locking also restricts editability when the document is small. Furthermore, support for fine-grained locking would have to be explicitly built into the application.

Finally, the pessimistic model is ill-suited for operation in environments with unreliable connectivity.

2.2 Edit-based

The edit-based approach is also simple. It relies on capturing all user actions and mirroring them across the network to other users. Algorithms based on Operation Transformation[1] are currently popular for implementing edit-based collaborative systems.

Obtaining a snapshot of the state is usually trivial, but capturing edits is a different matter altogether. A practical challenge with edit-based synchronization is that *all* user actions must be captured. Obvious ones include typing, but edits such as cut, paste, drag, drop, replacements and autocorrect must also be caught. The richness of modern user interfaces can make this problematic, especially within a browser-based environment.

Any failure during edit passing results in a fork. Since each edit changes the location of subsequent edits, one lost edit may cause subsequent edits to be applied incorrectly, thus increasing the gap between the two versions. This is further complicated by the best-effort nature of most networking systems. If a packet gets lost or significantly delayed, the system must be able to recover gracefully.

Edit-based collaborative systems are not naturally convergent.

2.3 Three-way merges

Three-way merges are found in Subversion,[12] the Mjølner Project,[10] Google Docs[6] and many other products. An overview of the process is shown in Figure 1:

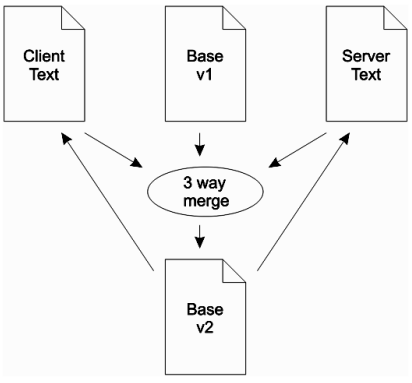

Figure 1: Three-way merge.

1. The client sends the contents of the document to the server.
2. The server performs a three-way merge to extract the user's changes and merge them with changes from other users.
3. The server sends a new copy of the document to the client.

If the user made any changes to the document during the time this synchronization was in flight, the client is forced to throw the newly received version away and try again later.[3] This is a half-duplex system: as long as one is typing, no changes are arriving. Shortly after one stops typing, the input from other users is integrated and either appears, or else a dialog pops up to let one know that there was a collision.

This system could be compared to an automobile with a windshield which becomes opaque while driving. Look at the road ahead, then drive blindly for a bit, then stop and look again. Major collisions become commonplace when everyone else on the road has the same type of "look xor drive" cars.

Server-side three-way merges do not scale well when attempting real-time collaboration across a network with latency.

3. DIFFERENTIAL SYNCHRONIZATION OVERVIEW

DS is a symmetrical algorithm employing an unending cycle of background difference (diff) and patch operations. There is no requirement that "the chickens stop moving so we can count them" which plagues server-side three-way merges.

Figure 2 is an idealized data flow diagram for DS. It assumes two documents (misleadingly called Client Text and Server Text) which are located on the same computer with no network.

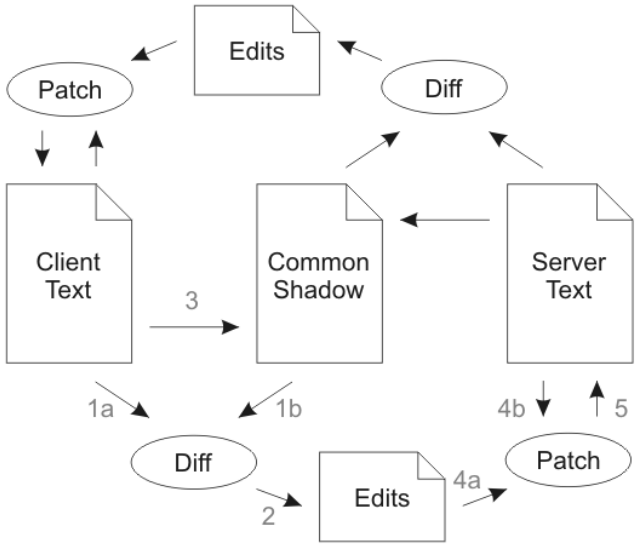

Figure 2: Differential Synchronization without a network.

The following walk-through starts with Client Text, Common Shadow and Server Text all being equal. Client Text and Server

[3] The three-way merges in version control systems (such as Subversion) usually happen on the clients, not the server. However this leads to the same issue, merely in reverse. If the server version updates while the client is merging, the client's merge will not be accepted.

Text may be edited at any time. The goal is to keep these two texts as close as possible with each other at all times.

1. Client Text is diffed against the Common Shadow.
2. This returns a list of edits which have been performed on Client Text.
3. Client Text is copied over to Common Shadow. This copy must be identical to the value of Client Text in step 1, so in a multi-threaded environment a snapshot of the text should have been taken.
4. The edits are applied to Server Text on a best-effort basis.
5. Server Text is updated with the result of the patch. Steps 4 and 5 must be atomic, but they do not have to be blocking; they may be repeated until Server Text stays still long enough.

The process now repeats symmetrically in the other direction. This time the Common Shadow is the same as Client Text was in the previous half of the synchronization, so the resulting diff will return modifications made to Server Text, not the result of the patch in step 5.

The enabling feature is that the patch algorithm is fuzzy, meaning patches may be applied even if the document has changed. Thus if the client has typed a few keystrokes in the time that the synchronization took to complete, the patches from the server are likely to have enough recognizable context that they may still be applied successfully. However, if some or all of the patches fail in step 4, they will automatically show up negatively in the following diff and will be patched out of the Client Text. Here's an example of actual data flow.

a. Client Text, Common Shadow and Server Text start out with the same string: "Macs had the original point and click UI."

b. Client Text is edited (by the user) to say: "Macintoshes had the original point and click interface." (edits underlined)

c. The Diff in step 1 returns the following two edits[4]:

```
@@ -1,11 +1,18 @@
 Mac
+intoshe
 s had th
@@ -35,7 +42,14 @@
 ick
-UI
+interface
 .
```

d. Common Shadow is updated to also say: "Macintoshes had the original point and click interface."

e. Meanwhile Server Text has been edited (by another user) to say: "Smith & Wesson had the original point and click UI." (edits underlined)

f. In step 4 both edits are patched onto Server Text. The first edit fails since the context has changed too much to insert "intoshe" anywhere meaningful. The second edit succeeds perfectly since the context matches.

g. Step 5 results in a Server Text which says: "Smith & Wesson had the original point and click interface."

h. Now the reverse process starts. First the Diff compares Server Text with Common Shadow and returns the following edit:

```
@@ -1,15 +1,18 @@
-Macintoshes
+Smith & Wesson
 had
```

i. Finally this patch is applied to Client Text, thus backing out the failed "Macs" → "Macintoshes" edit and replacing it with "Smith & Wesson". The "UI" → "interface" edit is left untouched. Any changes which have been made to Client Text in the mean time will be patched around and incorporated into the next synchronization cycle.

4. DUAL SHADOW METHOD

The method described above is the simplest form of differential synchronization, but it will not work on client-server systems since the Common Shadow is, well, common. In order to execute on two systems, the shadow needs to be split in two and updated separately. Conceptually the dual shadow method shown in Figure 3 is the same algorithm as the simpler version presented in Figure 2.

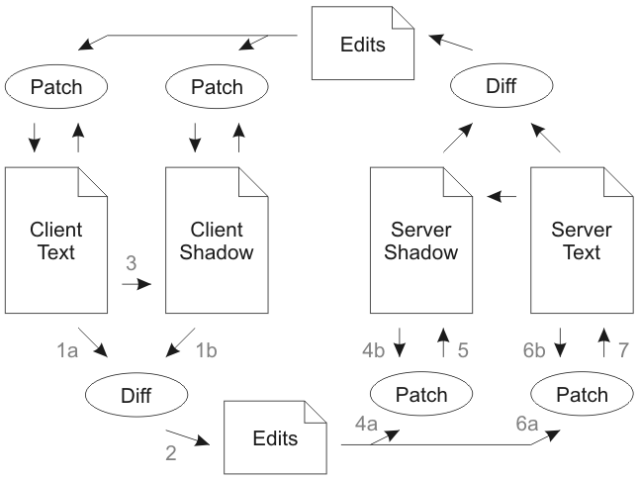

Figure 3: Differential Synchronization with shadows.

Client Text and Server Shadow (or symmetrically Server Text and Client Shadow) must be absolutely identical after every half of the synchronization. This should be the case since "(v1 Diff v2) Patch v1 == v2". Thus assuming the system starts in a consistent state, it should remain in a consistent state. Note that the patches on the shadows should fit perfectly, thus they may be fragile patches, whereas the patches on the texts are best-effort fuzzy patches.

However, on a network with best-effort delivery, nothing is guaranteed. Therefore a simple checksum of Client Shadow ought to be sent along with the Edits and compared to Server Shadow after the patches have been applied. If the checksum fails to match, then something went wrong and one side or the other must transmit the whole body of the text to get the two parties back in sync. This will result in data loss equal to one synchronization cycle.

[4] Edits are shown in the standard Unidiff format, but character-based instead of line-based. Lines starting with '@@' contain the expected location for the following edit, lines starting with '+' are insertions, lines starting with '-' are deletions, and the rest is unchanging context.

5. GUARANTEED DELIVERY METHOD

In the event of a transitory network failure, an outbound or a return packet may get lost. In this case the client might stop synchronizing for a while until the connection times out. When the connection is restored on the following synchronization, the shadows will be out of sync which requires a transmission of the full text to get back in sync. This will destroy all changes since the previous successful synchronization. If this form of data-loss is unacceptable, a further refinement shown in Figure 4 adds guaranteed delivery.

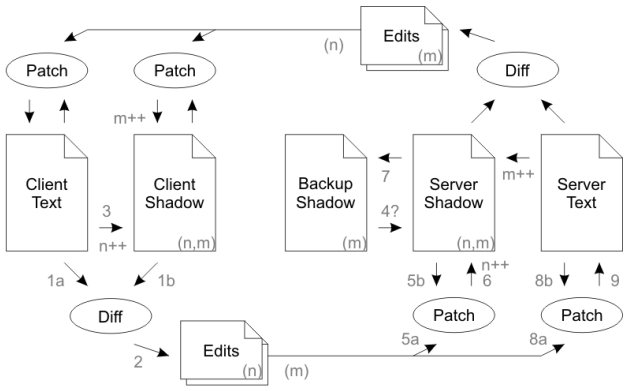

Figure 4: Differential Synchronization with guaranteed delivery.

In a nutshell: Normal operation works identically to the Dual System Method described above. However in the case of packet loss, the edits are queued up in a stack and are retransmitted to the remote party on every sync until the remote party returns an acknowledgment of receipt. The server keeps two copies of the shadow, "Server Shadow" is the most up to date copy, and "Backup Shadow" is the previous version for use in the event that the previous transmission was not received.

Normal operation: Client Text is changed by the user. A Diff is computed between Client Text and Client Shadow to obtain a set of edits which were made by the user. These edits are tagged with a client version number ('n') relating to the version of Client Shadow they were created from. Client Shadow is updated to reflect the current value of Client Text, and the client version number is incremented. The edits are sent to the server along with the client's acknowledgment of the current server version number ('m') from the previous connection. The server's Server Shadow should match both the provided client version number and the provided server version number. The server patches the edits onto Server Shadow, increments the client version number of Server Shadow and takes a backup of Server Shadow into Backup Shadow. Finally the server then patches the edits onto Server Text. The process then repeats symmetrically from the server to the client, with the exception that the client doesn't take a backup shadow. During the return communication the server will inform the client that it received the edits for version 'n', whereupon the client will delete edits 'n' from the stack of edits to send.

Duplicate packet[5]: The client appears to send Edits 'n' to the server twice. The first communication is processed normally and the response sent. Server Shadow's 'n' is incremented. The second communication contains an 'n' smaller than the 'n' recorded on Server Shadow. The server has no interest in edits it has already processed, so does nothing and sends back a normal response.

Lost outbound packet: The client sends Edits 'n' to the server. The server never receives it. The server never acknowledges receipt of the edit. The client leaves the edits in the outbound stack. After the connection times out, the client takes another diff, updates the 'n' again, and sends both sets of edits to the server. The stack of edits transmitted keeps increasing until the server eventually responds with acknowledgment that it got a certain version.

Lost return packet: The client sends Edits 'n' to the server. The server receives it, but the response is lost. The client leaves the edits in the outbound stack. After the connection times out, the client takes another diff, updates the 'n' again, and sends both sets of edits to the server. The server observes that the server version number 'm' which the client is sending does not match the server version number on Server Shadow. But both server and client version numbers do match the Backup Shadow. This indicates that the previous response must have been lost. Therefore the server deletes its edit stack and copies the Backup Shadow into Shadow Text (step 4)[6]. The server throws away the first edit because it already processed (same as a duplicate packet). The normal workflow is restored: the server applies the second edit, then computes and transmits a fresh diff to the client.

Out of order packet: The server appears to lose a packet, and one (or both) of the lost packet scenarios is played out. Then the lost packet arrives, and the duplicate packet scenario is played out.

Data corruption in memory or network: There are too many potential failure points to list,[14] however if the shadow checksums become out of sync, or one side's version number skips into the future, the system will reinitialize itself. This will result in data loss for one side, but it will never result in an infinite loop of polling.

5.1 Asymmetry

An obvious question is that given the otherwise perfect symmetry between client and server, why does the server have a Backup Shadow whereas the client does not? The source of this asymmetry is the asymmetrical nature of the connections. In a web-based client-server configuration, the client is the only entity which can initiate a connection. Depending on data losses, there are only three possible outcomes: 1) client sends data which is lost before reaching the server, 2) client sends data to server, but server's response is lost before reaching client, 3) client and server complete a successful round-trip. Notably missing is the possibility that the client's data is lost but the server's data is received. Every time the server sends information to the client, that implies a successful connection must have been established

[5] If using TCP/IP, duplicate and out of order packets should theoretically be impossible. However experience shows that there are a lot of routers and proxies on the Internet which take shortcuts and make mistakes.

[6] An alternative strategy would be to use the diffs in the edit stack to reverse patch the shadow back to the required version. This obviates the need for the backup shadow. While saving storage space, this is somewhat more computationally complex.

from the client to the server. Thus the server cannot get into a situation where it repeatedly sends packets to the client which don't arrive — while not obtaining any packets from the client.

The client could implement a Backup Shadow, but it would never get used when run on a web-based client-server architecture. For symmetrical architectures (e.g. peer-to-peer or server-to-server) where either side can initiate a connection to the other, then a Backup Shadow would be required on both sides.

6. TOPOLOGY

The above diagrams demonstrate synchronization between two parties, either a user and a server, or a pair of users. Figure 5 illustrates how the same synchronization strategy can be multiplied to service any number of additional clients in a server-centric network. The Server Text for each synchronization loop is common with all the other loops. When Client 1 changes his document, Server Text is updated upon the next synchronization cycle, and those changes are passed on to all other clients on the following cycle.

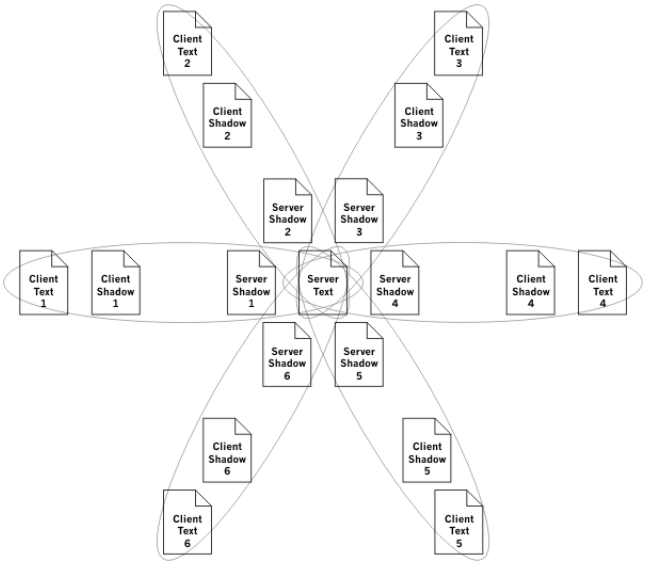

Figure 5: Six client, one server synchronization network.

Scalability may become an issue as the number of clients increase. Diff and patch can be expensive operations, thus a server may become overloaded. There are two simple methods of distributing the system onto multiple servers.

One method illustrated in Figure 6 is to separate the database from the algorithm. Thus one database would service any number of load-balanced servers. A client could hit any server, and as long as the view of the shared database is identical across all servers, the system remains consistent.

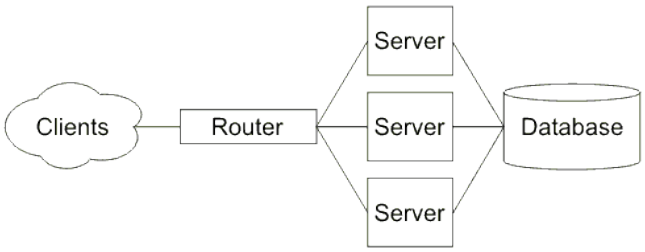

Figure 6: Many servers, single database.

Another method illustrated in Figure 7 is to introduce a server-to-server topology. In the diagram below, the clients are divided equally between two servers and the two servers are linked to each other with exactly the same type of connection as between the servers and the clients. Additional servers may be added seamlessly whenever capacity is exceeded. Servers may only be removed when all their clients depart and they only have a single connection to another server.

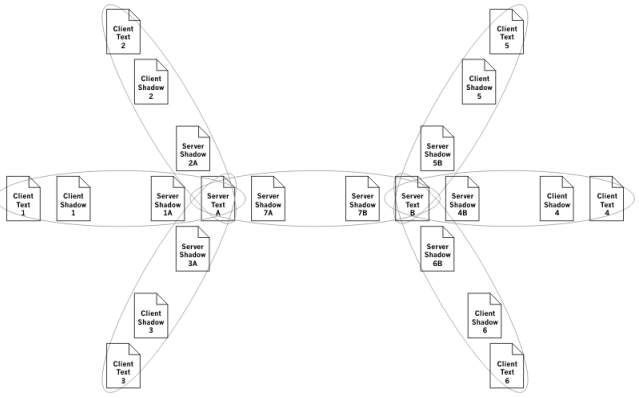

Figure 7: Six client, two server synchronization network.

As the network expands, a potential problem is latency. Each link might synchronize every five seconds (see section 8). Thus it would take a change from Client 1 up to fifteen seconds to appear for Client 4. As latency increases, so does the potential for non-trivial collisions. Accordingly it is important to avoid a long chain of servers; a balanced tree offers the shortest path between clients, and thus the least latency.

Latency may also be reduced by significantly increasing the synchronization frequency between servers. If the servers are located next to each other, then there is no bandwidth cost in synchronizing several times a second.

7. DIFF AND PATCH

All the examples in this paper have shown synchronization of plain text. DS can handle any content (plain text, rich text, bitmaps, vector graphics, etc) as long as a difference algorithm and a fuzzy patch algorithm are available for the content.

As the only computationally expensive components of DS, improving the efficiency of these algorithms dramatically improves the responsiveness and scalability of the system. Likewise, improving the accuracy of these algorithms greatly reduces the number and severity of collisions.

7.1 Diff

The diff operation is fulfilling two very different roles within the synchronization cycle. The first is to update Server Shadow with the current content of Client Text and Client Shadow. The result should make all three texts identical. This is a simple task which could use any form of synchronization; diff, delta edits[7] or even transmission of the full text. The second operation is more of a challenge: updating Server Text with the changes made to Client Text. Server Text may have changed in the mean time, which means that the diff must be semantically meaningful.

For instance, if the word "cat" was deleted and replaced with "hag", then technically one could think of it as the replacement of the first and third letters, with the second letter being preserved. This would be the minimal diff.

```
Client Text:   The cat is here.
Client Shadow: The hag is here.
Minimal Diff:  The ~~c~~h~~at~~g is here.
Semantic Diff: The ~~cat~~hag is here.
```

But this was not the semantic intent of the user. The user changed the word, not two letters. The fact that 'a' was the same in both words was completely coincidental. This distinction matters because if in the mean time another user changed the server's text from "cat" to "cut", the result when applying the first user's patch should be either "hag" (client wins) or "cut" (server wins), but certainly not "hug" (merged differences). An algorithm must be used to expand minimal diffs into semantically meaningful diffs. [2]

Another issue with diff is its lack of scalability. The leading plain-text diff algorithm is O(nd) where n is the length of the text and d is the length of the changes.[11] Clearly this does not scale for long documents or large edits. Fortunately the general-purpose diff algorithm can be wrapped inside a number of shortcuts which typically obviate the need to run this expensive algorithm at all.[2] The following three shortcuts apply specifically to plain-text differencing, but variations of them may be applicable to other content.

7.1.1 Equality

The overwhelming majority of diffs in DS are comparing two identical texts with each other. Detecting this special case can be done with a single == operation.

7.1.2 Common Prefix/Suffix

If there is any commonality at all between the texts, it is likely that they will share a common substring at the start and/or the end.

```
Text1: The cat.
Text2: The black cat.
```

After removing the common prefix and suffix in this example one gets "" (the empty string) and "black " respectively. Identifying the common prefix or suffix can be done in linear time with a simple loop, or by comparing substrings in a binary search. Figure 8 shows a test of the running time of these two algorithms as n increases geometrically.

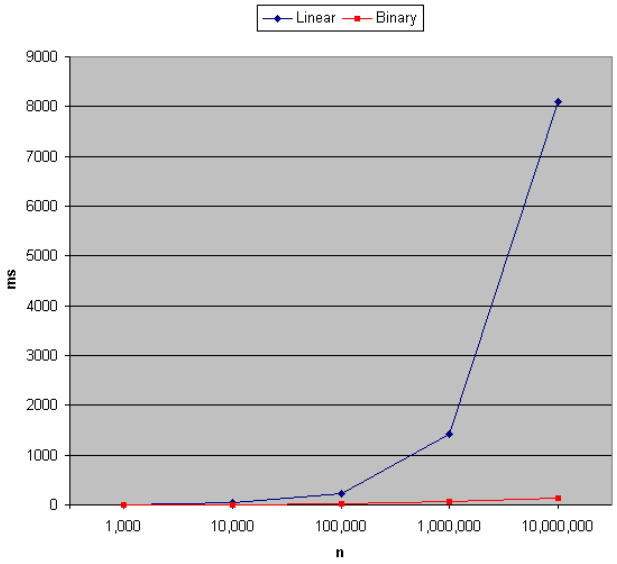

Figure 8: Performance of Linear vs. Binary search algorithms.

The linear search scales at O(n) as expected. The binary search scales at O(log n). This result is counter-intuitive given that the binary search ought to be O(n log n) when one considers that an equality operation itself is likely to be O(n). However within the context of a high-level language, the performance of low-level operations such as "==" effectively becomes O(1). The data graphed above comes from timings of a JavaScript implementation; tests in Python show the same pattern. Obviously in a low-level language such as C the linear search would be superior to the binary search.

7.1.3 Singular Insertion/Deletion

After the common prefix and suffix are trimmed off, it is very common that one or the other strings is empty. The above example resulted in "" (the empty string) and "black " which clearly represents an insertion of "black ". In the case where the second string is the empty string, then the operation is a deletion. Neither case requires running the general-purpose difference algorithm.

The net result of these three shortcuts is that if a diff is executed frequently enough to catch each change individually (where one change can be an insertion or a deletion of arbitrary length), then the general-purpose difference algorithm is never executed and performance becomes O(log n) for languages where string equality is a constant time operation and O(n) for languages where string equality is linear.

Of course there remains the pathological case of an instantaneous change of the whole document. Consider selecting all the existing text then pasting new text from the clipboard. To guard against this case the difference algorithm can be equipped with a timeout which if reached will cause it to simply return a deletion of the old text and an insertion of the new text. While this may not be the minimal difference, it is likely in such cases to be the semantic intent of the user and thus preferred in the context of DS.

7.2 Patch

The patch operation is just as critical to the operation of the system. This system requires that patches be applied in and

around text which may not exactly match the expected text. Furthermore, patches must be applied 'delicately', taking care not to overwrite changes which do not need to be overwritten.

Patch must look at two (potentially conflicting) variables when attempting to find the correct place to make an insertion. The first is to find text with the smallest Levenshtein distance between the expected text (based on the context of the patch) and the actual text.[5] The second is to find a location reasonably close to the expected location of the patch. It is probably more correct to apply a patch onto a near-match at the expected location than to a perfect match at the other end of the document.

The Bitap algorithm offers a remarkably efficient method of locating near-matches in plain-text.[15] Once the best match location is identified, a diff can be run against the expected text and the actual text, thus creating an accurate translation matrix of indices from one text to the other.[3] The index of the patch may then be updated, and finally the patch may be applied.

User complaints identified one necessary exception to the requirement of delicate patching. In cases where the shared content is entirely numeric or part of a limited set of allowable values, the patches should not be merged. Consider the following text merge:

```
Base:  cat
User1: Cat
User2: cats
Merge: Cats
```

While this is an ideal merge of the users' combined intentions, the same merge when done with numbers is clearly not correct:

```
Base:  145
User1: 845
User2: 1459
Merge: 8459
```

A similar case is found when collaborating a field that has enumerated types, such as a dropdown list of the days of the week ('Monturday' is not usually acceptable). In cases of non-mergable content, patching should simply be skipped, and a "last user wins" approach taken.

7.2.1 Handling Collisions

Implementations of DS must consider the consequences of patch errors. This is a usability issue, not an algorithmic one. Some applications may choose synchronize very frequently and quietly drop patches which don't fit. The advantage of this option is that in the event of a collision there are at least two users whose attention is currently focused on the offending area, and either one of them can correct the content which they typed in the past second or two.

Other applications may choose to synchronize less frequently and require user interaction on failed patches. This might be in the form of inline annotations that the users can resolve when they wish. The advantage of this option is that users' content is never dropped or mangled.

8. ADAPTIVE TIMING

The frequency of each client's update cycle is a key factor in determining the responsiveness of the system. Insufficiently frequent updates result in more computationally expensive diff and patch operations, major edit collisions, merge failures, and frustration when attempting to interact with other users. Overly frequent updates result in higher network traffic and increased system load.

An advantage of the Guaranteed Delivery Method described above is that it decouples the differencing operation from the network transmission. Diffs can be taken at frequent intervals (to conserve CPU resources), added to the edit stack, then transmitted in batches at a slower rate (to conserve network resources).

An adaptive system can continuously modify the network synchronization frequency for each client based on current activity. Hard-coded upper and lower limits would be defined to keep the cycle within a reasonable range (e.g. 1 second and 10 seconds respectively). User activity and remote activity would both decrease the time between updates (e.g. halving the period). Sending and receiving an empty update would increase the time between updates (e.g. increasing the period by one second). This adaptive timing automatically tunes the update frequency so that each client gradually backs off when activity is low, and quickly reengages when activity is high.

9. FUTURE WORK

The fuzzy patch operation is actually a simple implementation of a three-way merge. The two branches (Server Text and the post-update Server Shadow) and the base version (the pre-update Server Shadow) are all available on the server. Thus one could choose instead to use one of the many other three-way merge algorithms which are available.[8] Naturally the same opportunity applies on the client side.

One limitation of DS as described here is that only one synchronization packet may be in flight at any given time. This would be a problem if there was very significant latency in the connection. An example would be a client on Mars and a server on Earth. A half hour for the round trip at the speed of light is unavoidable, however it would be better to send a continuous stream of updates in each direction, not waiting for the reply to arrive. The algorithm does not currently support this feature.

Another avenue for exploration would be to keep track of which user was responsible for which edits. Currently the edits from all users are blended together on the server, making attribution difficult. Untangling this blend would allow incoming edits to be visually attributed to specific users, as well as potentially allowing rollbacks of individual contributions and other features available in source control systems.

10. CONCLUSIONS

Differential Synchronization builds upon existing difference and patch algorithms to produce a robust collaborative platform. The use of differences eliminates the need to detect edit events directly and makes the system naturally convergent. The guaranteed synchronization method solves both the problem of network failures and the problem of batching small (more efficient) edits into one connection.

Differential Synchronization has proven itself to work extremely well as implemented by MobWrite and compatible systems. It is impressively accommodating of multiple users who are working on the same text. Tests consistently show that MobWrite's technical scalability far exceeds the point where social scalability breaks down. The limits of social scalability depend on the nature

of the collaborators, the size of the document and the nature of the tasks being performed. For example a dozen coworkers simultaneously fixing OCR errors in a large document works well, whereas a single anonymous web user on an open wiki can render it unusable. By contrast, the limit of technical scalability on an existing single-server implementation has been load tested as lying in the 100 edits per second range.

A finding from user observation is that some form of communication channel should be available to collaborators. This may take the form of instant messaging, the telephone or just talking over a cubical wall. Failure to provide an out-of-band channel will result in users subverting the system to create an in-band channel. Documents end up littered with temporary user-to-user chat messages which they often forget to clean up afterwards.

11. ACKNOWLEDGEMENTS

Tancred Lindholm provided extensive guidance on improving this paper for which I am deeply grateful.

12. REFERENCES

[1] Ellis, C.A. and Gibbs, S.J. 1989. Concurrency control in groupware systems. In *International Conference on Management of Data (SIGMOD)*. Portland, OR.

[2] Fraser, N. *Diff Strategies*. Retrieved April 13 2009: http://neil.fraser.name/writing/diff/

[3] Fraser, N. *Fuzzy Patch*. Retrieved April 13 2009: http://neil.fraser.name/writing/patch/

[4] Fraser, N. *MobWrite*. Retrieved April 13 2009: http://code.google.com/p/google-mobwrite/

[5] Levenshtein V.I., *Binary codes capable of correcting deletions, insertions, and reversals.* Soviet Physics Doklady 10 (1966): 707–710.

[6] Lindholm, T. A three-way merge for XML documents. In *Symposium on Document Engineering (DocEng) 2004*, Milwaukee, WI, October 2004.

[7] MacDonald, J. *File system support for delta compression*. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[8] Mens, T., *A State-of-the-Art Survey on Software Merging*, IEEE Trans. Software Eng., vol. 28, no. 5, pp. 449-462, May 2002.

[9] Microsoft Knowledge Base: *File Locking in Master Documents*, Article ID 176313. Retrieved April 13 2009: http://support.microsoft.com/default.aspx?scid=kb;EN-US;176313

[10] Minör, S. and Magnusson, B., A Model for Semi-(a)Synchronous Collaborative Editing, *Proceedings of ECSCW'93, Third European Conference on Computer Supported Cooperative Work*, Milano, Kluwer Academic Publishers, 1993.

[11] Myers, E.: An O(ND) Difference Algorithm and its Variations. Algorithmica 1(2): 251-266 (1986).

[12] Pilato, C.M., Collins-Sussman, B. and Fitzpatrick, B.W. *Version Control with Subversion, 2nd Edition*. O'Reilly, Sebastopol, 2008, 121-122.

[13] Saito, Y. and Shapiro, M., Optimistic Replication, *ACM Computing Surveys (CSUR)*, vol 37, issue 1, pp. 42-81, March 2005.

[14] Stone, J. and Partridge, C., *When The CRC and TCP Checksum Disagree*, Proceedings of 2000 SIGCOMM, Stockholm, Sweden, August 28 - September 1, 2000.

[15] Wu, S., and Manber, U., *Fast text searching with errors*, Tech. Rep. TR-91-11. Department of Computer Science, University of Arizona., Tucson, AZ, June 1991.