No Lost or Laden Updates in a Collaborative Web Application

Jake Smola

December 13, 2017

1 Introduction

In the context of distributed systems, the "no lost updates" problem concerns the preservation of all users' contributions to a shared resource given multiple users may modify the resource simultaneously. This paper describes our solution to the no lost updates problem within a collaborative web application called "El Capitan".

El Capitan is a prototype cyber attack planning application that allows red teams to document cyber attack tactics, techniques, and procedures and coordinate their attacks in a common setting. We began the project in August of 2017 with certain functional priorities in mind; as a result, our recent addition of no lost updates logic was heavily influenced by the following aspects of the current design: (1) El Capitan is a single-page application where the primary client interface is fully contained within the web browser; (2) the architecture includes a single database server that stores persistent data and a single, one-thread web server that hosts the web page logic and queries the database on behalf of the clients; (3) the client interface consists of a visual, directed acyclic graph of data-containing nodes that can be added as children of existing nodes, removed, or modified by users; (4) the client can only have outstanding (uncommitted) changes to one node at a time[1]

These characteristics differentiate El Capitan from typical applications seeking to prevent lost updates, such as version control systems like git. For instance, version control systems generally operate in parallel with a file system that constitutes the primary client interface for writes and reads. However, our solution is restricted to the the context of a web application where the user's primary interface is embedded within the application itself and is thus divorced from the user's local file system. In section 3 we will show that the characteristics of El Capitan actually allow us to simplify our implementation of no lost updates.

[1]This is for two reasons: (1) we consider the web browser to be an unstable platform where user work may be lost often due to various user, server, or network errors; (2) most web browsers have limited local storage (often only 10 MB) in which to store large transient data.

2 Related Work

At a high-level, preventing lost updates can be broken down into two steps:

1. Detecting conflicts between diverging user updates to shared resources
2. Acting to preserve the effects of said updates.

Many of the tricks version control systems utilize in implementing these two steps mirror or are derived from contemporary work in distributed computing.

On the topic of conflict detection, one of the simplest solutions involves the use of version vectors, as originally described in [4]. When implemented, each data source is associated with a version vector that tracks the (Lamport) time at which the data was modified at each replica.

Tra is an alternative to version vectors that also includes in each vector coordinate the (Lamport) time at which the data was synchronized [1]. This addition afford a more expressive notation with which a replica can ascertain just how much of a file's history of updates another replica is aware of. Vector time pairs thus facilitate more efficient synchronization or resolution than do version vectors alone.

Some existing storage protocols incorporate derivatives of versionining in so-called dependency checks when detecting conflicts. *Bayou* write procedure calls take as input the requesting replica's expected result to determine whether or not concurrent writes have impacted the target data, or in

other words, whether or not the version being modified by the replica corresponds to the version stored at the other end [5]. Put operations in Amazon's *Dynamo* key-value store also utilize a form of dependency check by taking as input the requesting replica's *context*. This context contains vector clock information that likewise provides data version insights and supports causality and conflict determinations [2].

With respect to conflict resolution, the two-way and three-way merge are commonly used to reconcile conflicting changes to a resource(s). The common two-way merge model takes two resources and identifies identical and distinct portions of each resource and alerts the requesting client(s) of the conflicts for manual resolution. The three-way merge model takes two resources and a third (ancestor) resource from which the two resources are derived and improves on the output of the two-way merge by possibly making some merge decisions automatically on behalf of the clients. The three-way merge is able to do this by using the ancestor resource to determine the precise changes each client made since the ancestor version. In this case, changes that do not conflict can be applied without asking the client to take action. For this reason, we employ *diff3*, an open-source, three-way merge library [3], for conflict resolution since it simplifies the user's task.

3 Method

We will now describe our no lost updates additions to El Capitan. There are seven actions a user can take in El Capitan that induce a write operation:

1. Add a node
2. Add an edge
3. Remove a node
4. Remove an edge
5. Update node data
6. Import a graph
7. Change a node color

However, most of these actions require little effort and by design, overwrite the effects of the previous action of the same kind. For instance, a node can only have one color, and an edge either exists or it does not. Thus, we only provide no lost update support for the following pairs of concurrent behaviors on **nodes**:

1. Remove-Update (Forgotten Update)
2. Update-Update (Conflicting Update)
3. Add-Add or Update-Add (Conflicting Add)

It is important to note that in our case the web server is the sole source of queries on the database and operates on a single thread, thus blocking simultaneous client transactions on the database (rest API calls). Thus each action will either commit or abort before another action is considered.

We will now describe our no lost update solution for each of the above cases.

3.1 Forgotten Update

This scenario occurs when client A attempts to update a node that client B had removed prior to the latest synchronization of client A. These two actions cannot be merged and so we must choose which action is committed. Based on the induced hierarchy of nodes and data, we chose to commit the node removal since this is analogous to removing a directory in which another user is modifying a file. Furthermore, a client could remove an entire path of the nodes making updates to lower tier nodes irresolvable. Still, to prevent loss, we allow the updating client to recover by prompting them with a warning stating the node being updated no longer exists, and that the client should save their changes externally to the web application if they wish to preserve their work.

3.2 Conflicting Update

This scenario occurs when two users concurrently update the same node. A merged outcome is possible in this case and so we can construct an update protocol to detect conflicts and if necessary merge changes.

We wish to reliably detect conflicts with as little overhead as possible. In section 2 we described version vectors, which are among the cheapest methods we encountered for detecting conflicts. We thus incorporated version vectors into our first prototype design. However, we soon found their organic implementation to still require more state than was

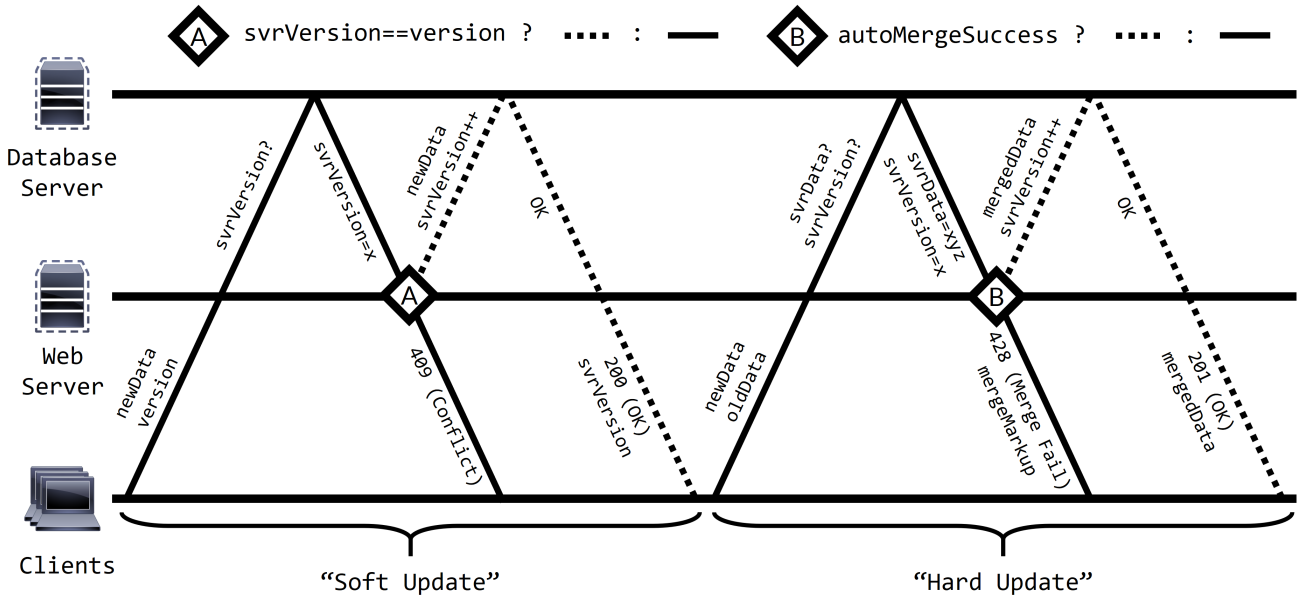

Figure 1: El Capitan update protocol.

necessary for our problem. In the typical setting where version vectors are employed, each replica can communicate with every other replica. This requires each replica to have oversight of every other replica's version and hence, each replica maintains a vector in which each coordinate corresponds to one replica. In our case, we have a centralized server managing all commits and clients cannot communicate amongst themselves. We can therefore simplify the version vector paradigm by having the server alone maintain a version vector. However, this still requires significant state on behalf of the server as it must track a potentially highly dynamic pool of clients. We can thus off-load server state and have each client maintain the version of each node in their possession. During an update, the client can then provide their version information to the server to facilitate conflict detection. Our precise implementation uses Lamport timestamps as versions. New nodes are assigned version 0 and each time a node is updated by any client, its version is incremented. When a client reads node date (on initial page load or refresh), it downloads the most recent version alongside each node. In the non-byzantine setting, with these invariants in place, the server can immediately accept updates to nodes whose database version matches the version passed by the client during update, as this signifies the client is updating the same version of the data they had originally read. We call this type of update a *soft update*. A *soft update* succeeds if no conflict is detected and fails otherwise.

If a client's updated data conflicts with data already committed by another client, the server will attempt to reconcile the conflict with as little input from the human user as possible by using a three-way merge. However, the the three-way merge requires more data than is provided in the *soft update* sequence. The process requires a version of the data being modified that causally precedes both of the conflicting versions. As in the case of git, the server can store the history of each node to identify this common "ancestor" when resolving conflicts. However, this would be extremely space-intensive. Since we already trust the client to provide the appropriate version during *soft update*s, we can also off-load this responsibility to the client by having the client send to the server the original version of the data that the client had read prior to writing. Unlike the server, the client would not need the entire node history because it only needs to find the immediate ancestor from which its new data is derived, and not an earlier ancestor it shares with any other, potentially severely outdated client. Since the client does not update its local view until a change has committed, the client does not actually need to store any additional data. Additionally, the data associated with the client's most recent read is the most recent ancestor of both conflicting versions and thus

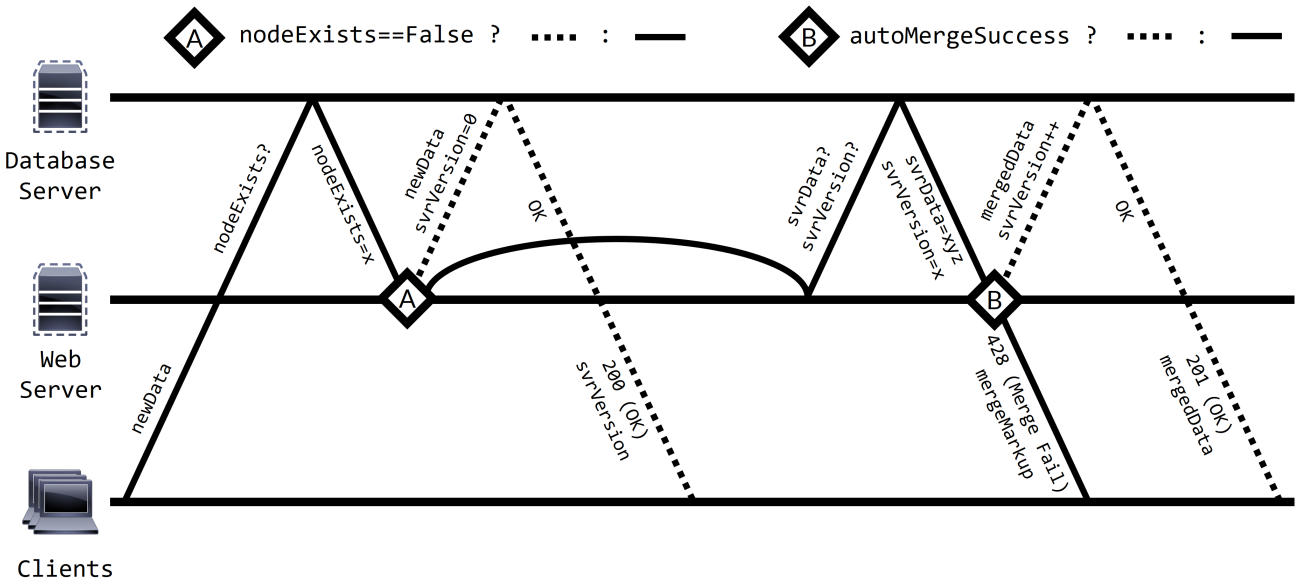

Figure 2: El Capitan add protocol.

the *optimal* ancestor for the merge. This is the case because there is no earlier ancestor with respect to the client's new data, and because the client can only read *committed* data from the data base, so any changes to this data between some client's read and subsequent write operations must also be derived from (merged with) the original data read by the client.

In the case of a conflict during a *soft update*, the server thus requests the original data from the updating client. To prevent the server from having to store potentially unbounded state, client has to include the new data already sent during the *soft update* again in its response. The server can consequently run *diff3* to compute a three way merge. If the merge succeeded, the server can update the database. If the update could not be automatically reconciled, the server sends the partially merged data back to the client for human reconciliation. Unfortunately, given the properties of our problem, this partially merged data is only stored ephemerally in the user's browser session and thus must be reconciled within the same session to prevent data loss. Users are encouraged to store the data persistently on local disk in the event they cannot reconcile the data during the session. We call the update procedure in the case of a conflict (failed *soft update*) a *hard update*.

El Capitan's refined update protocol is depicted in figure 1.

3.3 Conflicting Add

The conflicting add scenario occurs when a client adds a new node that already exists (shares a node label and a parent node with an existing node). El Capitan handles these conflicts similarly to the way it handles updates. The only differences include: (1) when the client requests to add the node, the node in question is not yet recognized by the server so it does not have a version and the server instead verifies whether or not the node already exists; (2) in the case of conflict, the server can immediately attempt a *hard update* of the existing node since no ancestor data exists.

El Capitan's refined add protocol is depicted in figure 2.

4 Evaluation

We evaluate our update protocol in terms of *ACID* properties, latency as compared to an intuitive alternative implementation, and space complexity requirements. We forego analysis of the add protocol as it is derived from the update protocol and only makes improvements on its workflow.

4.1 ACID Properties

Neo4j, the back-end database system we utilize already provides ACID guarantees and so our analysis is limited to assessing whether or not any server

or client data dependencies violate these guarantees outside of *Neo4j*'s workflow.

In terms of *atomicity*, an update cannot be committed unless a query is dispatched to *Neo4j*; because no additional data is mutated after the web server queries the database on a particular update, the atomicity of our update shares fate with each *Neo4j* query. Because *Neo4j* queries are atomic, our updates are atomic.

Regarding *consistency*, we know an update can fail at one of three locations: the client, the web server, or the database. If the update fails at the server or the client, then it will have aborted since no query would have been sent to the database. If an update fails at the database, then *Neo4j* is responsible for guaranteeing consistency with a rollback to a previous of the data.

In terms of *isolation*, we know the server operates on a single thread and does not begin processing additional updates until a *soft update* succeeds or the subsequent *hard update* terminates. Therefore, simultaneous update transactions are blocked and concurrent data operations on the same node cannot conflict.

Finally, we guarantee *durability* since all update transactions are assumed aborted until committed. If an update transaction is committed, it means Neo4j has guaranteed it is committed. Otherwise, the transaction cannot possibly be committed because the durable state can only be stored in Neo4j, which has not committed the transaction.

4.2 Latency

We consider an intuitive alternative to our update protocol (baseline) to consist entirely of *hard updates* in order to eschew multiple back-and-forth messages. We fix the following variables to conduct our analysis, assuming an enterprise LAN environment connects the servers and clients: (1) network bandwidth, propagation delay, and queuing between all assets is uniform and fixed; (2) the sum of end-to-end propagation and queuing delay for all pairs of assets is 100 μs (3) HTTP response frames are 200 bytes; (4) node data is fixed at 500 bytes (before and after updates); (5) the data rate between all assets is fixed at 10 Mb/s; (6) HTTP/TCP hand-shakes are negligible and HTTP keep-alive is active; (7) the web server and database server are co-resident on the same hardware and thus network traffic between these two assets is negligible; (8)server-side computation time is negligible.

We rely on an ideal latency model where d represents delay and total latency $L = d_{\text{propagation}} + d_{\text{queueing}} + d_{\text{packetization}}$. Let L_s, L_f, and L_h respectively represent the latency of a successful *soft update*, the late latency of a failed *soft update*, and the latency of a *hard update*. Let p represent the proportion of *soft updates* that succeed. Note that $L_f = L_s + L_h$.

Under our ideal model, our protocol's latency is $L = pL_s + (1-p)L_f$, which we compare to the baseline latency $L' = pL_h$.

Plugging in our fixed variables, we get:

$$L_s = 2(1\text{e}^{-4}) + \frac{8(700+200)}{10\text{e}^6} = 9.2\text{e}^{-4}$$
$$L_h = 2(1\text{e}^{-4}) + \frac{8(1200+200)}{10\text{e}^6} = 1.32\text{e}^{-3}$$
$$L_f = L_s + L_h = 2.24\text{e}^{-}3$$

Plugging these values back into L and L', we find that $L \leq L'$ when $p \geq 0.\bar{8}4$.

This means our update protocol incurs as much or lesser latency than the baseline when soft updates succeed at least 85% of the time. We believe actual El Capitan usage will meet this criteria as the intended users of this application consist of small teams accessing a large amount of data.

4.3 Space Complexity

It is difficult to compare El Capitan to any existing application of a similar nature, however, as compared to the previous version of El Capitan, the new version's space requirements differ marginally. Across the database, the web server, and the clients, each asset only needs to maintain an additional key-value pair for each node, consisting of the integer node version.

5 Conclusion

Given the ACID, space and latency properties of our implementation, we believe our no lost updates protocols are both reliable and efficient.

In the future, will be augmenting our merge procedure to support word-by-word merging instead of *diff3*'s native character-by-character merging. This

change will help users merging HTML documents involving primarily English text.

We would have liked to also incorporate *authenticated* client-provided ancestor data and versions during conflict detection to support byzantine scenarios. However, given El Capitan's exclusively internal corporate use case, we have not prioritized this addition.

References

[1] Cox, R., Josephson, W. File Synchronization with Vector Time Pairs. *Technical Report MIT-CSAILTR-2005-014*, MIT, 2005.

[2] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W. Dynamo: Amazons highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM Press New York, NY, USA, pp. 205–220, 2007.

[3] Housel, B. Node-diff3. Available: https://www.npmjs.com/package/node-diff3.

[4] Parker, D.S., Popek, G.J., Rudisin, G., Soughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, 1983.

[5] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., and Hauser, C.H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM Press New York, NY, USA, pp. 172—182, 1995.