Real Time Co-editing of High Vertex Count Geometries Using OpenLayers and CRDTs - a Performance Analysis

Hrvoje Matijević [1], Nikola Kranjčić [1], Vlado Cetl [1], Saša Vranić [2]

[1] Department of Geodesy and Geomatics, University North, Jurja Križanića 31b, Varaždin, Croatia - (hmatijevic, nkranjcic, vcetl)@unin.hr
[2] Geoweb - svranic@geoweb.hr

Keywords: Strong eventual consistency - SEC, Conflict-free replicated data types - CRDT, geospatial co-editing, Real time GIS, performance under load.

Abstract

When manipulating high vertex count geometries, real-time collaborative editing of geospatial data presents significant performance challenges. This paper investigates the performance of a CRDT (Conflict-free Replicated Data Type) based real-time geospatial co-editor implemented in JavaScript using OpenLayers and Reference CRDTs. It extends previous work by evaluating system responsiveness under increasing data complexity and concurrent user load. To this end, large polygons with 100K, 200K, and 300K vertices were co-edited by up to 60 concurrent users across varying hardware platforms. Key performance bottlenecks were identified in the GUI coupling and CRDT integration mechanisms. While the core CRDT mechanisms remained performant, GUI limitations emerged as the primary constraint at higher vertex counts. A modified downstream processing approach was implemented to mitigate the detected GUI limitations. The results suggest that, despite some GUI limitations, CRDT-based architectures are viable for real-time co-editing of high vertex count geometries even under increased active user counts.

1. Introduction

Real-time GIS has become an essential tool in various domains, including Volunteered Geographic Information (VGI) and disaster management. One of the critical topics in real-time GIS is concurrency control (Sun and Li 2016). In the geospatial domain, concurrency has traditionally been controlled using optimistic or pessimistic models (i.e. versioning and locking, respectively). In the domain of distributed databases, a standard consistency model, called strong consistency, ensures that a set of distributed databases behave as if they were a single database. However, enforcing strong consistency can introduce bottlenecks and lags, and requires significant hardware resources and time to implement.

To address these drawbacks, a more relaxed approach called strong eventual consistency (SEC) (Gomes et al., 2017) has been developed in the domain of real-time text co-editing. Unlike strong consistency, SEC lets each site edit its local copy of the data without any restrictions and replicate all the updates to all other sites, which, upon reception, apply them on their local data. Temporary local inconsistencies are allowed between the participating sites, but it is guaranteed that, once all sites have received the same set of updates, they will be in the same state (i.e., they will converge).

It has recently been shown that SEC model i.e. its instantiation, CRDT (Commutative Replicated Data Type) technology (Shapiro et al., 2011) can also be used for the task of geospatial co-editing. Within the research (Matijević et al., 2024), an experimental real-time geospatial co-editor (source code referenced in the original paper) has been developed and tested. The implementation uses OpenLayers (OL) (OSGEO, 2007) on the graphical user interface (GUI) and a small, not heavily optimized but complete and correct JavaScript CRDT library called Reference CRDTs (Gentle, 2023). The research showed that, when applied to the co-editing of geospatial geometry in its native form, standard CRDT conflict resolution mechanics exhibit some issues. As an attempt to address these issues, the authors developed an advanced operation generation technique named “tentative operations”. This technique allows for the operations to be generated over the most recent session-wide state of the data, which in effect highly reduces concurrency and provides a “geometry aware” conflict resolution.

Real-time co-editors generally aim to provide an excellent user experience of the system, with correct handling of conflicts being one of its important aspects. However, besides correct handling of conflicts, the system also has to be responsive. The responsiveness of real-time co-editors depends not only on the efficiency of the underlying business logic but also on the efficiency of the GUI itself. Especially in the case of geospatial data manipulation, both the GUI and the business logic will be additionally stressed by increasing the vertex count of geometries being co-edited. Since within the original research (Matijević et al., 2024) the tests were performed using polygons with very low vertex count (several hundreds), it remained unknown how a CRDT based geospatial co-editor would behave when much larger geometries (e.g. hundreds of thousands of vertices) are co-edited.

Within this research, we therefore investigate the impact that the increase of vertex count has on the overall performance of the system, which in turn can hinder responsiveness. We reused the existing implementation from (Matijević et al., 2024) and introduced some modifications to better address the efficiency of execution of several key mechanisms as well as to achieve the ability to time their execution. The source code of the modified implementation and the data used for the experiments is available on GitHub (Matijević et al., 2025). Instead of focusing on the performance of CRDT mechanisms only, such as in (Briot et al., 2016), we observed the behaviour of the complete system. To stress the system, we created three polygons with 100K, 200K and 300K vertexes and conducted

real co-editing sessions over those polygons with an increasing number of concurrent participants (up to 60). During test sessions, we timed the execution of the key mechanisms used in the implementation.

Within this research, we are not interested in finding the fastest possible CRDT implementation for geospatial co-editing but in understanding the major performance considerations and relative ratios between the major parts of such an implementation. There exist both standard as well as novel, hybrid CRDT-OT (operational transformation) libraries that are significantly faster than Reference CRDTs (Gentle and Kleppmann, 2025).

The structure of the remainder of the paper is as follows. In section 2 we provide a light introduction to the main concepts of CRDTs in general and define a general framework that the rest of the paper builds upon. In section 3 we provide a more specific elaboration of the key mechanisms that heavily influence the overall performance of CRDT based co-editors. Section 4 describes the setup and the actual testing performed within the research. Section 5 discusses the test results and section 6 concludes the paper.

2. Background and Setting the Stage

Following, a brief and high-level description of main CRDT mechanics is given. Readers interested in mode theoretical details are directed to the original paper (Matijević et al., 2024) and the literature referenced there.

The primary concept with fixed-size identifier CRDTs is their data structure's view and model space. The view space is always kept identical to the GUI's resource. It does not contain deleted elements so the position of an element in GUI's resource is always identical to its position in the CRDT's data structure view space. The model space contains deleted elements (called tombstones). Consider a CRDT data structure holding four points (P1-P4). Initially, the model and the view are identical. When an update (e.g. P2 replaced by its new version via a delete+insert) is done the model and the view are no longer identical, with positions of points P3 and P4 in the model and in the view being different (Figure 1).

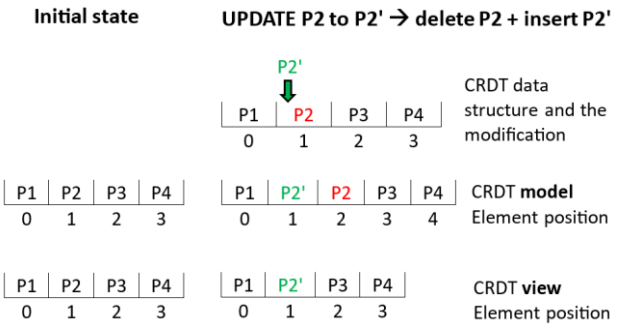

Figure 1. Behaviour of CRDT model and view after integration of an operation.

Keeping tombstones is needed to be able to consistently integrate elements across sites. In various situations (explained later) conversion of an element's position between CRDT's data structure view and model space is needed.

Fixed size identifier based CRDTs integrate operations using element's left neighbour. Locally (on the site that created the element), the new element is inserted into local CRDT data structure based on its position in the view, directly. Then, its immediate left neighbour's id is stored into the element's metadata. When the element arrives on a remote site, first its left neighbour is found by identifier in that site's local data structure. Then the remote element is positioned somewhere to the right of its left element's position, depending on whether other elements with the identical left neighbour exist.

Typically, CRDT implementations are made of two distinct phases. The so-called upstream phase generates CRDT operations upon GUI edits and integrates the elements locally. It must convert a position-based operation specification provided by the GUI into identifier-based operation (a CRDT element). This is done by view-to-model mapping. The upstream phase doesn't need to update the GUI since the operation is generated following the edit executed on the GUI, which is already rendered (Figure 2).

The so-called downstream phase integrates the received remote operations. Unlike the upstream phase, the downstream phase must update the GUI. To update the GUI, it must convert the identifier-based operations back to position-based update specification by model-to-view mapping or must do the full model-to-view conversion to generate a completely new state of the resource to be rendered on the GUI. This will depend on how the coupling between the GUI and the underlying CRDT mechanisms is implemented.

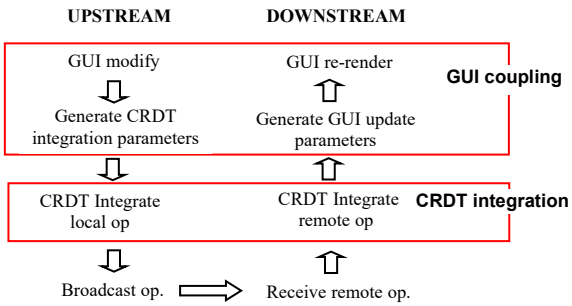

Figure 2. Upstream and downstream phases of CRDT based co-editor.

The two phases can be observed in two different aspects. First aspect is the process of detecting and preparing the GUI updates to be submitted to the CRDT mechanism on the upstream phase. Also, in the downstream phase, the results of the remote updates must be prepared and rendered on the GUI. We call this GUI coupling. The second aspect of the two phases is standard CRDT integration mechanics.

Next, we analyse the performance hot spots of GUI coupling and CRDT integration mechanisms. This is specifically done for the Reference CRDTs although all the concepts should apply to any list based CRDTs.

3. Performance "Hot Spots"

List based CRDT implementations typical have two performance "hot spots". If not carefully implemented, those can result in an overall degradation of performance. We first elaborate on the hot spots within the GUI coupling and then on the hot spots in the actual CRDT integration algorithm.

3.1 Upstream GUI Coupling

General purpose GUIs typically do not use identifiers that could be used to generate CRDT elements' integration parameters

directly. Thus, an implementation will typically need to take the position-based specification of the GUI's update (i.e. update's specification in the view) and then generate the identifier-based CRDT data element's integration parameters. This is done by view-to-model conversion. In our specific case, OL generates and exposes position-based update specification in its drag segments, thus performing the first part of the upstream GUI coupling process. Using this, and by executing the view-to-model mapping the CRDT element's integration parameters are determined. Then, when the user completes the edit (by finishing the dragging phase), the final position for the edit is determined and the CRDT element can be finalized (its content defined) (Figure 3).

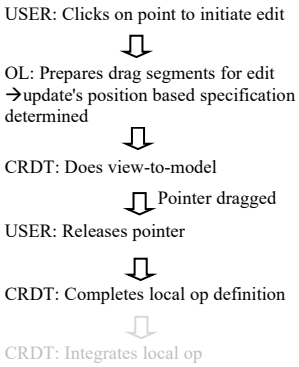

Figure 3. Upstream GUI coupling.

There are two computationally heavy "hot spots" here. First is determining the position-based operation specification. As said, this is done by OL and exposed in its "drag segments".

The second hot spot in the upstream GUI coupling, the view-to-model conversion, is in Reference CRDTs done by simply counting tombstones to the left of the element's position in view. This is an inefficient, brute force implementation. A possible approach to address the inefficiency of view-to-model implementation is to introduce an additional identifier index (Briot et al., 2016).

3.2 Downstream GUI coupling

The downstream phase of the GUI coupling must, given a new state of the CRDT data structure created by integrating a remote element, generate the parameters for the GUI re-render. Since identifiers stored within the CRDT element cannot be used for this, identifier-based position will need to be converted back to position based specification, hence requiring a round of model-to-view conversion (Figure 4). The model-to-view conversion filters out the tombstones in order to generate the "visible" state of the resource, which is to be rendered on the GUI.

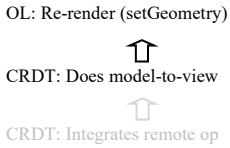

Figure 4. Downstream GUI coupling.

To do the model-to-view conversion, implementations that keep CRDT elements in a linked list must traverse all elements by references and materialize only non-tombstone ones. Different to that, Reference CRDTs keep elements in an array, hence already correctly ordered. Still, filtering out the tombstones does take time.

The actual OL redraw is in our specific case executed by replacing the entire current geometry with the new geometry for each remote operation (using OL's setGeometry method). As we show later this easily gets inefficient.

3.3 CRDT Integration

During integrating local operations (i.e. in the upstream phase), the view-to-model mapping (done within the GUI coupling part), suffices to correctly position the local CRDT element on its final position in the local CRDT data structure. Its integration parameters (e.g. its left and possibly right neighbour's ids) are determined as its direct neighbours. No searching is needed here. The new element then enters standard integration algorithm execution which is identical for local and remote elements. In the local case, since the local element's neighbours' positions are already known, the integration algorithm does not need to run the costly find-by-identifier mechanism to find them. Hence, the performance of the upstream phase depends almost exclusively on the performance of the view-to-model conversion mechanism.

This is different in the downstream phase. When integrating a remote CRDT element, the integration algorithm must first find its original neighbours in the local data structure by identifiers. Then, it might need to do multiple iterations before it finds the correct position for the remote element to be inserted into. During each iteration, additional find-by-identifier calls might be needed.

To find an element by identifier in the CRDT data structure, a brute force implementation can scan it sequentially from the start until it finds it. This, given n elements in the data structure, can in the worst-case result in O(n) time complexity. Since the find-by-identifier mechanism sometimes needs to be called multiple times per element being integrated, this is a suboptimal situation. Each call to a brute force find-by-identifier mechanism when the element is positioned towards the end of the data structure will introduce overhead to the overall performance.

Various speed-up techniques, including hash-tables (Lv et al., 2019) with O(1) time complexity and balanced trees (Nicolaescu et al., 2016) with O(log n) time complexity are used to speed up the find-by-identifier mechanism. An alternative technique, borrowed from the original YJS implementation (Jahns 2016) is also used by Reference CRDTs. Each client remembers the last position it inserted an element into, so when a new element is to be inserted then first the neighbourhood of the last inserted element is searched. Since often subsequent operations occur near the previous one (with text but also with geometry editing), this simple heuristic can cheaply (i.e. without introducing additional data structures) reduce the find-by-identifier performance overhead. Nevertheless, if a user moves to a more distant part of the geometry, this approach will fail and a brute force full scan will need to be done. In Reference to CRDTs this technique is called range index so we reuse it here. We upgraded the Reference CRDTs implementation to keep the range index for each remote site separately.

The second part of the integration mechanism is the actual low-level insertion of the new element into the data structure. Linked list-based implementations will do it in O(1) time. When inserting an element into an array, the array-based implementation (as is Reference CRDTs) needs to shift all the array elements to the right of the newly inserted position, which

can be costly. Nevertheless, modern JavaScript engines exert the ability to insert into arrays in near constant time.

Following the analysis from this subsection, we identified the need to separately monitor the performance of CRDT integration in the upstream and downstream phases. For integrations in the downstream phase, besides the execution time, we logged range index misses. Both integration times (local and remote) include the time needed to insert the element into the CRDT data structure (i.e. JavaScript array).

4. Tests

4.1 Test Setup

The original implementation uses an in-buffer for processing of remote operations (downstream). Its original purpose was, coupled with the tentative operation mechanism, to enable delayed integration of remote operations that are received during the period that the user is dragging. However, in the original implementation, if a remote operation arrives while the user is not dragging, it is integrated and GUI redrawn immediately (orange downstream flow option on Figure 5). During the original testing, such a setup presented no problems since the target geometry had a low vertex count and GUI redraws were very fast and not noticeable. Also, it was considered that seeing in real-time what other users are doing on the target geometry presented a benefit for user experience.

During preliminary testing within this research, we noticed that with high vertex count geometries, OL takes a long time to redraw while in "modify" state. This, in turn, made view manipulation (panning and zooming) from laggy to virtually impossible to do (depending on the vertex count and hardware used). Therefore, we modified the implementation to enable delayed integration and GUI redraw only after completing local edits. We achieved this by reusing the existing tentative operations mechanism. Now, instead of only buffering the remote operations received during dragging, all remote operations are buffered. All buffered remote operations are integrated in a single burst after the user completes the local edit (green downstream flow option on Figure 5). As a consequence, OL redraw is only executed once for multiple remote operations. Note that when OL is not in "modify" state, redraws are done much faster and do not hinder view manipulation, so we implemented the delay mechanism to only works when OL is in "modify" mode.

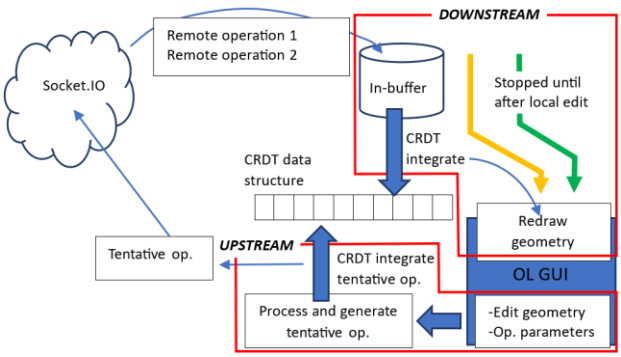

Figure 5. System architecture with original (orange) and modified (green) downstream processing flow.

For the purpose of testing, we used two different low-cost desktop hardware platforms. The first, low-end (LE) platform (with a 1164 single-core score in geekbench 6) was based on a 3.9 Ghz capable AMD Ryzen 2600 CPU, with 16 Gb of DDR4 ram and a Nvidia GT 740 GPU. The second, high-end (HE) platform (with a 1875 single-core score in geekbench 6) was based on an up to 4.2 Ghz capable AMD Ryzen 5500, with 16 Gb DDR4 ram and a Nvidia 1030 GPU. For reference, at the time of writing of this paper the current top single-core performing CPU was 5.8 Ghz capable Intel Core i9-13900KS with a score of 3129 in single core performance (Geekbench Browser 2025). Both platforms were running MS Windows 11 Pro 24H2. All the tests were done within Google Chrome Version 136.0.7103.49 (Official Build) (64-bit).

We created three polygons using a small lake outline merged to three different CORINE polygons made out of 100K (Figure 5), 200K and 300K points.

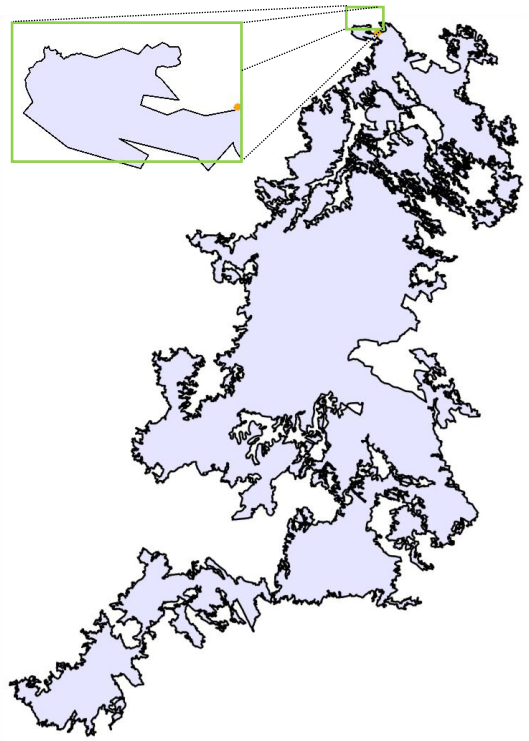

Figure 6. 100K test polygon with target area enlarged (orange point is polygon origin).

Thus, the tester could always work on the identical polygon (lake outline), disregarding its very large rest. We deliberately merged the lake outline to CORINE polygons to create the worst-case situation where the lake polygon was placed at the very end of the large polygon (the coordinates for the polygon are ordered in the clockwise direction). This means that each range index miss resulted in full scanning of 100K, 200K and 300K arrays when running the find-by-identifier mechanism.

We ran six test editing sessions, three on each of the two platforms. In each of the three sessions per platform the same tester edited a 100K, 200K and 300K polygon.

To be able to simulate many active clients, we implemented a simple automatic operation generation capability on the client. Automatic operation generation can be configured to randomly generate operations with varying parameters. The configurable parameters are: target portion of the geometry, the number of consecutive local operations on the current position, the delay between two consecutive operations and the probability for different operation type to be generated. The parameters were setup as follows:

- Target geometry portion: last third of the geometry

- Nr. of ops on current position: 1-10
- Delay between generating two consecutive local ops: 1.5-3.5 seconds
- Operation type probabilities: insert 40%, update 40%, delete 20%

Such a configuration forced the automatic clients to generate operations within the last third of the geometry (the more difficult part), to change position after at most 10 operations generated (causing index misses) and to prefer inserts and updates over deletes (as can in most cases be expected). We ran automatic clients on two mid-range, high core-count computers. The tests were done within a low-latency LAN.

Within each session the tester was asked to do the identical flow of editing, in two phases (Figure 7). Both of the phases started from the northeast corner where most of updating needs to be done, followed by the regions where mostly inserts are needed.

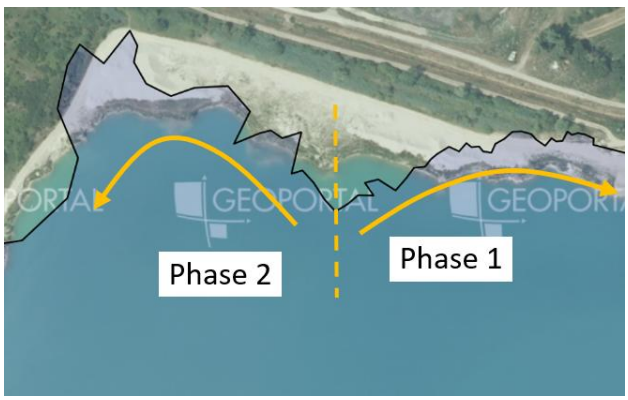

Figure 7. Visualization of the two editing phases used during test sessions.

The active automatic client load distributions across phases is shown in Table 1:

Editing phase	Duration (sec)	Active automatic clients
Inactive	60	15
1	60	15
	60	30
2	60	45
	60	60

Table 1. Distribution of active clients over the session duration.

We measured various execution times using JavaScript's performance.now() method. First, before each of the target code block a call to performance.now() method recorded current time. Then, after target block completed computations, we again call the method and calculate the difference between the two, thus producing the specific code block's execution time. Since we expected ranges in tens of milliseconds, all execution times were rounded to a tenth of millisecond.

4.2 Test Results

We first provide numbers of local operations that the tester generated within each session, the numbers of session wide operations generated by automatic clients and the numbers of range index misses for each test session (Table 2).

Plat.	Metric	100K	200K	300K
HE	Local ops generated	119	101	92
	Automatic remote ops processed on test client	3950	3907	3984
	Range index misses	794	748	757
LE	Local ops executed	113	95	79
	Automatic remote ops processed on test client	3935	3917	3972
	Range index misses	775	802	783

Table 2. Basic session parameters.

Within all six test session the test client and all the automatic clients ended up with the identical state (a session wide convergence achieved).

On the subjective user experience side, the tester reported significant lags in OL snapping functionality in both 300K session. However, in both 100K sessions, the tester reported no noticeable lag in OL snapping. The 200K sessions exerted some lag but such that it did not significantly reduce user experience. The tester did not report any jitters or lags in view manipulation even in the 300K sessions.

Next, we first elaborate on measured CRDT integration times and follow up by GUI coupling times.

4.2.1 CRDT integration times

In the original implementation, non-concurrent local (upstream) CRDT integrations did not need the service of range index since no calls to find-by-identifier needed to be done. Such operations are correctly positioned already after the view-to-model conversion. In the changed environment with the tentative operations mechanism having to execute virtually each time (given a constant influx of remote operations), local integrations heavily relied on range index. Indexed local CRDT integration times remained constantly below 1ms for all six sessions. Unindexed local integration times were at the level approximately corresponding to that of unindexed remote integrations. Nevertheless, since we configured the range index to operate with a generously wide range (100) and since the tester did not change the part of the geometry being edited often, only an insignificant number of range index misses occurred (3-5 per session).

Indexed remote integration times were also in all cases below 1ms. Unindexed remote integrations occurred more frequently than local ones since automatic clients changed position more often and as a rule moved to distant parts of the target geometry. We calculated average execution times and standard deviations (SD) needed for integrating unindexed remote operations using the execution times achieved in inactive phase for each session (Table 3).

Plat.	Metric	100K	200K	300K
HE	Average	1.1	1.9	2.5
	SD	0.3	0.4	0.3
LE	Average	1.3	2.3	3.5
	SD	0.3	0.4	0.5

Table 3. Unindexed remote op execution times (ms).

As expected, the highest average unindexed remote integration time of approximately 2ms (along with a significant standard deviation of 3ms) was measured on LE platform at 300K

polygon. Note that all CRDT integration times include the time needed to insert the actual element into the target JavaScript array.

4.2.2 GUI coupling - upstream

We first look at the upstream phase of GUI coupling. The view-to-model part of the upstream GUI coupling topped at around 3ms on LE platform at 300K polygon (Table 4).

Plat.	Metric	100K	200K	300K
HE	Average	0.6	1.1	1.5
	SD	0.1	0.2	0.3
LE	Average	0.8	1.9	2.6
	SD	0.2	0.4	0.3

Table 4. View-to-model times (ms).

The reader is at this point reminded that, to do model-to-view conversion, the operation's position in view must be available. As stated earlier, this is provided by OL. Besides providing operation's position in view, OL enables real-time tracking and snapping of pointer in real-time. To provide all this, OL must employ spatial indexing and performance optimized data structures internally. Maintaining all this uses CPU resources, hence with vertex heavy geometries it will cause a performance penalty when such structures must be initiated or refreshed.

4.2.3 GUI coupling - downstream

Next, we look at the downstream phase of GUI coupling. Time needed to integrate a single remote operation depends on the operation type and whether range index hit or missed. However, since operations are not executed immediately as they arrive on site but in bursts, various combinations of operation types and range index hit/miss rates per burst will result in different execution times over operation counts. If the user works in a slower pace or even pauses, number of operations to be integrated in a burst will grow accordingly. Within the experiment we asked the tester to work in his usual pace but constantly and without pauses. Nevertheless, while repositioning himself to a different editing location (e.g. switching from phase 1 to phase 2) more operations will be buffered.

The unsystematic behaviour of burst execution times over session active automatic client loads is well demonstrated by the varying burst execution times over a number of operations in a burst for the entire active part of a session. The two examples, for 300K polygon on both platforms are depicted by the two graphs on Figures 8 and 9.

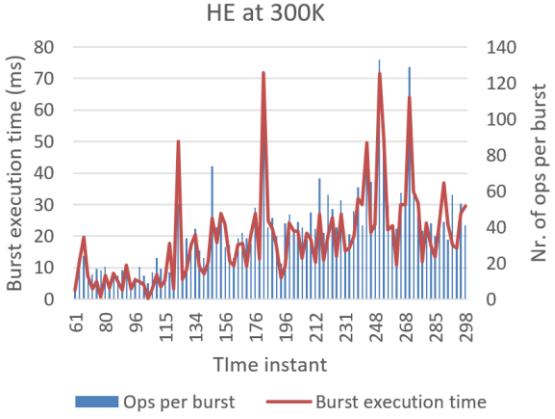

Figure 8. Burst execution times over number of ops per burst for HE platform at 300K.

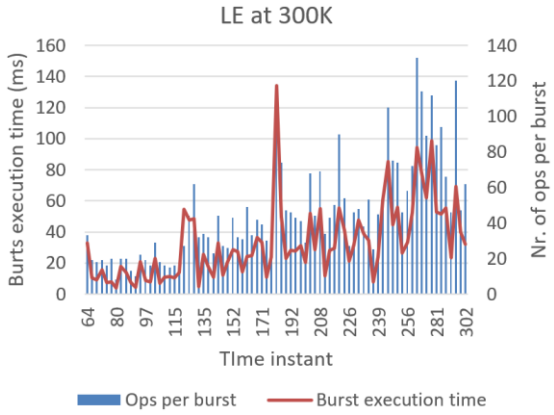

Figure 9. Burst execution times over number of ops per burst for LE platform at 300K.

Given such an unpredictable configuration, instead of calculating averages, to get the insight into the expectable execution times we calculated 90th percentiles (exclusive, using MS Excel's percentile.exc) of execution times per burst for each of the six test sessions and for each active automatic client load (Figure 10).

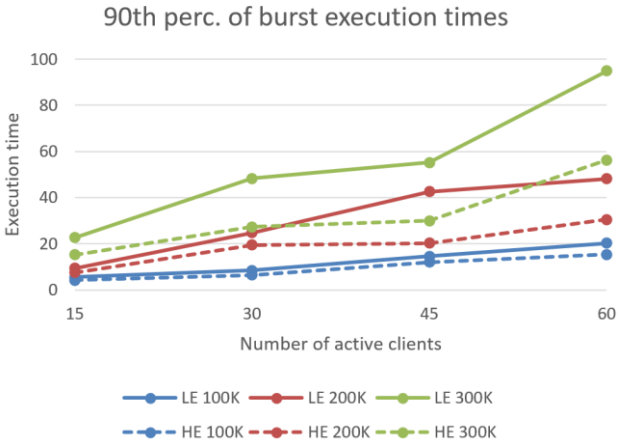

Figure 10. 90th percentile of burst execution times per session and per active clients load.

The data shows that at 100K both platforms stayed within the 25ms limit. The HE platform was able to hold below 50ms limit up until the highest active client loads even at 300K. Even the LE platform managed to stay below 50ms limit during complete 200K session, but crossed it in the 2nd half of 300K session.

In terms of the overall GUI redraw process, there are two aspects to consider. First, CRDT implementation must prepare the data to be drawn. Then GUI must do the actual redrawing. The preparation of the data to be redrawn is done by model-to-view conversion. As can be seen, at 300K both platforms required 10 and more milliseconds to prepare the data for redrawing (Table 5).

Plat.	Metric	100K	200K	300K
HE	Average	2.8	6.0	8.6
	SD	0.7	0.9	1.6
LE	Average	3.9	7.6	12.5
	SD	1.0	1.2	4.3

Table 5. Model-to-view times (ms).

We now look at the actual OL redrawing performance. Redrawing when OL is not in "modify" state ranged from 5 to 10ms for 100K case and topped at almost 30ms for 300K case (on LE platform) (Figure 11).

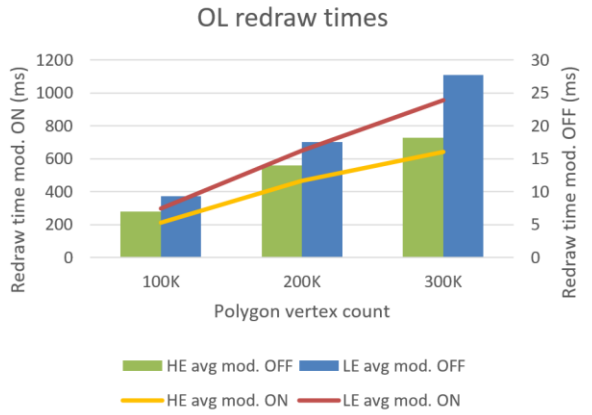

Figure 11. OL redraw times ("modify" state on and off).

While this could still be considered acceptable, redrawing when OL is in "modify" state ranged in unacceptable 200-1000ms. Given those ranges, the first part of the redraw process (model-to-view conversion) can be considered irrelevant. As stated earlier, the very high overall redraw times when OL is in "modify" state was the primary reason we had to reconfigure the system to execute remote operations in bursts, thus reducing the number of OL redraws in "modify" state to an acceptable level.

5. Discussion

In any human-computer interaction scenario, it is desirable to keep the system response times (SRT) below a threshold that would make the computer seem unresponsive to the user. However, defining this threshold is not a straightforward task and can vary based on the type of work, age of the user, experience of the user to only name a few. Still, usually the delay of below 100ms is generally accepted as an acceptable threshold for responsiveness (Attig et al., 2017). We therefore divide the 100ms into two equal parts, and consider that each of the two engines (CRDT and GUI) has 50ms at its disposal.

The upstream phase is less interesting since CRDT part executes very fast and based on the test results has a lot of headroom to sustain geometries with even larger vertex counts. Furthermore, it depends heavily on the OL performance which we at this point cannot influence nor analyse.

In the downstream phase, HE platform exerted the ability to keep its CRDT times below the 50ms limit in 90% of cases (90th percentile) even at 300K polygon and 60 active session participants. The LE platform broke the limit earlier and topped at almost 100ms. Nevertheless, the important part to notice here is that, since the buffered operations are executed after the user completes his local edit, the performance penalty introduced by executing a burst of remote operations is not noticeable. This even holds in cases when the 50ms limit is heavily crossed (such as we had at 300K on LE platform). This is because at that point the user is focused on finding his next edit only moving the screen pointer towards it.

In terms of OL downstream performance, redraw times when in "modify" state are obviously a bottleneck in this particular case and cannot be implemented in full real-time. Thus, this had to be addressed by delaying redraws to the latest possible phase and executing once per multiple remote integrations. Identical as in the case of downstream CRDT integration, the lag introduced by redrawing does not hinder the user experience since it occurs only when the user has finished his local edit and is focused on the next one.

A deeper dive into OL internal engines and avoiding the use of setGeometry method could possibly fix real-time OL redraw performance in "modify" state. Still, since OL is based on JavaScript, possibly this could remain a hard limit. Alternatively, different geospatial GUI libraries (e.g. written in Rust and running in WASM) could provide better performance. An investigation of benefits of RUST over JavaScript can be found in (Wang et al., 2025) where CRDT technology is also used for geospatial collaborative work.

6. Conclusion

Traditional concurrency control techniques (locking and versioning) are, and most likely will prevail in the majority of geospatial co-editing scenarios. However, in situations where an uncontrollable number of co-editors need the ability to do ad-hoc, rapid work, geospatial co-editors based on CRDT technology offer several benefits. The system can be deployed and made operational very quickly, on minimal central hardware and the users will be able to do co-editing with no constraints. This has already been shown by previous research. However, CRDT based implementations do most or all of the heavy processing on clients. This means that client business logic needs to be able to provide the performance needed to ensure acceptable GUI latency on the target hardware platform.

Within this research, we have shown that the overhead introduced by the various aspects of CRDT technology, even when running in JavaScript, on the clients based on moderately capable hardware and in increased session participant ratios, does not result in significant system performance degradation. This even holds when the geometries being edited have high vertex counts, with 300K vertexes tested in practice. By reusing the tentative operations technique which delays the local integration of multiple buffered remote operations until the user completes his local edit, the negative effects of GUI lag are reduced to minimum.

In this particular case, the main performance bottleneck was found to be the GUI. This primarily relates to real-time snapping and redrawing of the target geometry when GUI is in the more demanding, "modify" state.

Although a production level implementation of a real-time web GIS co-editor will use some highly optimized CRDT framework and might want to modify the standard GUI to achieve a tighter coupling, the results from this research do provide some lessons learned and identify some of the hot spots to be carefully addressed in such a case.

Acknowledgements

This research is supported by the scientific project UNIN-TEH-25-1-7 Applications of Artificial Intelligence in Geomatics from the University North, Croatia.

References

Attig, C., Rauh, N., Franke, T., & Krems, J. F., 2017. System latency guidelines then and now–is zero latency really

considered necessary?. In Engineering Psychology and Cognitive Ergonomics: Cognition and Design: 14th International Conference, EPCE 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings, Part II 14 (pp. 3-14). Springer International Publishing.

Briot, L., Urso, P., & Shapiro, M., 2016. High responsiveness for group editing crdts. In Proceedings of the 2016 ACM International Conference on Supporting Group Work (pp. 51-60).

Gentle, J., 2023. Reference Crdts. https://github.com/josephg/reference-crdts (7 May 2025)

Gentle, J., Kleppmann, M., 2025. Collaborative Text Editing with Eg-walker: Better, Faster, Smaller. In Proceedings of the Twentieth European Conference on Computer Systems (pp. 311-328).

Geekbench Browser, 2025. Processor Benchmark Chart https://browser.geekbench.com/processor-benchmarks (7 May 2025)

Gomes, V.B.F.; Kleppmann, M.; Mulligan, D.P.; Beresford, A.R., 2017. Verifying strong eventual consistency in distributed systems. Proc. ACM Program. Lang. 2017, 1, 1–28.

Jahns, K., 2016. YJS. https://github.com/yjs/yjs (7 May 2025)

Lv, X.; He, F.; Yan, X.; Wu, Y.; Cheng, Y., 2019. Integrating selective undo of feature-based modeling operations for real-time collab-orative CAD systems. Futur. Gener. Comput. Syst. 2019, 100, 473–497. https://doi.org/10.1016/j.future.2019.05.021.

Matijević, H., Vranić, S., Kranjčić, N., Cetl, V., 2024. Real-Time Co-Editing of Geographic Features. ISPRS International Journal of Geo-Information 13, no. 12: 441. https://doi.org/10.3390/ijgi13120441

Matijević, H., Kranjčić, N., Cetl, V., Vranić, S., 2025. Geo-Coeditor https://github.com/HrvojeMatijevic/Geo-coeditor/tree/main/foss4g_mostar_2025

Nicolaescu, P.; Jahns, K.; Derntl, M.; Klamma, R., 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In Proceedings of the 2016 ACM International Conference on Supporting Group Work, Sanibel Island, FL, USA, 13–16 November 2016.

Shapiro, M.; Preguiça, N.M.; Baquero, C.; Zawirski, M., 2011. Convergent and Commutative Replicated Data Types. Bull. EATCS 2011, 104, 67–88.

Sun, Y., Li, S., 2016. Real-time collaborative GIS: A technological review. ISPRS J. Photogramm. Remote Sens. 2016, 115, 143–152, doi:https://doi.org/10.1016/j.isprsjprs.2015.09.011.

OSGEO, 2007. OpenLayers https://openlayers.org/ (7 May 2025)

Wang, B., Zhao, Q., Zeng, D., Yao, Y., Hu, C., Luo, N., 2025. Design and Development of a Local-First Collaborative 3D WebGIS Application for Mapping. ISPRS International Journal of Geo-Information, 14(4), 166.