ChatGPT

Insights into Yjs CRDT Structure and Protocol

1. Yjs CRDT Output: Log vs. Current State Representation

Yjs treats *edits* (operations) as first-class citizens, rather than treating the document state itself as the primary stored object [1]. In practice, a Yjs document's internal CRDT structure retains a record of all inserts and deletes (edits) that have been applied. The current document *state* is not stored as a simple snapshot; instead, it is constructed in memory on the fly by assembling all these edits [2]. In other words, Yjs follows an *event sourcing* approach – the persistent data is the sequence of edits, and the visible state is derived from replaying or integrating those edits in a specific order.

Because every insert or delete is preserved (though with some compression of consecutive inserts and truncation of deleted content), the internal structure is effectively a **log of operations** rather than a static state dump [3] [4]. Even when content is "removed," the corresponding insert entries remain as tombstones (with their content trimmed for efficiency) so that the intention and position of past operations are still recorded [5]. This design ensures that if two peers have the same set of edits, they will compute the same document state, fulfilling the CRDT convergence criteria.

Conclusion: The output from `verifyfilestructure.js` (which presumably dumps Yjs's internal data) represents a **hybrid** of history and state. It is not a mere snapshot of the final text, but rather the structured log of all edits (inserts and tombstones) that produce the current state. In essence, it's the complete set of operations (the "single source of truth" events) needed to reconstruct the document [6] [7]. This means the output includes the current content as well as metadata for conflict resolution and historical operations, reflecting Yjs's approach of permanent, conflict-free edit logs.

2. Yjs Message Types and Communication Format

Yjs uses a compact binary protocol to synchronize documents and to share presence (awareness) information. There are two main categories of WebSocket messages in Yjs: **document sync messages** and **awareness messages**. These are distinguished by a one-byte prefix in the binary message where `0` denotes a sync message and `1` denotes an awareness update [8]. Below is a breakdown of these message types and their purposes:

Message Type 0 – Full Resync or Heartbeat?

When the first byte of a message is `0`, it indicates a *sync protocol* message. Within the sync protocol, there are three sub-types (identified by the second byte) that correspond to different stages of synchronization [9] [10]:

- `[0,0]` **– SyncStep1:** This is essentially a *sync request*. It contains a state vector (a summary of which edits the sender already has) and asks the other party (often the server) for any edits that the sender is missing [11]. In practice, a client will send `[0,0]` upon connection (or after a disconnect/

reconnect) to ask for updates. If periodic full resyncs are enabled (via a `resyncInterval`), the client may also send a `[0,0]` message at intervals as a **full document resync request** [12] [13]. This is not a no-op heartbeat – it triggers the receiver to compute and send any missing document updates. However, if the document is already in sync, the response may be minimal or empty (essentially serving as a health check that all edits are received).

- `[0,1]` **– SyncStep2:** This is the reply to a SyncStep1. It includes all edits (inserts and deletes) that the requester is missing [10]. In effect, `[0,1]` can deliver a bulk of document updates – essentially a diff from the server's state to bring the client up to date. This often constitutes a "full document sync" for a newly connected client (or after a gap in connectivity). Once this exchange is done, client and server have the same set of edits.

- `[0,2]` **– Update:** This represents an *incremental update* message containing new edits (operations) that were produced by a client [14]. After initial sync, whenever a user makes an edit, the client broadcasts it as a `[0,2]` message. The server simply relays this to other peers. Each such message contains one or more insert/delete operations encoded in Yjs's binary format.

Heartbeat considerations: Yjs does not use a separate explicit heartbeat message for sync; instead, it relies on the awareness updates (type 1, see below) as well as the above resync mechanism if configured. Typically, the **awareness protocol** pings presence updates every ~30 seconds to avoid peers thinking each other have gone offline [15]. Thus, *message type 0* is not a trivial heartbeat ping – it carries state vectors or actual edits. If `resyncInterval` is disabled (which is the default in most cases [12]), a fully synced client will not keep sending `[0,0]` messages periodically. They occur mainly on initial connection or if an application explicitly triggers a resync. In summary, **message type 0 indicates a document sync exchange**, typically for full sync or diff sync purposes, rather than an empty heartbeat.

Awareness Messages – Presence Information

If the first byte of a message is `1`, it is an **awareness update** message [8]. The Yjs awareness protocol is a lightweight CRDT that broadcasts each user's presence and status to others [16]. **Awareness messages are solely about presence and user state**, not document content. They usually contain a small JSON-like state (encoded in binary) indicating things like the user's name, cursor position, color, or any application-defined ephemeral data for that user.

Awareness updates are used to answer "who is online and what are they doing?" For example, when a user joins a session, their client sends a type-1 message to announce "I am here" along with any details (e.g., name, avatar, current cursor position). Similarly, if a user moves their cursor or becomes idle, the client can broadcast an updated awareness message. When a user disconnects (or fails to send updates for a timeout period), peers will consider them offline and remove their presence state [15].

Importantly, awareness messages do **not** include document edits and do not affect the document's content. They are only to propagate *presence information (online status, cursors, etc.)* [17]. In Yjs's default WebSocket provider, the server simply relays these messages to all clients in the same room. Clients merge awareness updates using an internal awareness CRDT so that each client maintains a map of who's online and their last announced state [16]. In summary, **"awareness" in Yjs is purely about user presence and ephemeral collaborative context**, separate from the document's actual data.

Yjs Binary Update Format (Compact Chunks)

All Yjs sync messages (the `[0,1]` SyncStep2 and `[0,2]` update messages) carry document updates in a *highly compact binary format*. Rather than JSON or plain text diffs, Yjs encodes inserts and deletes as binary data (Uint8Arrays) to minimize size [18]. The format packs the essential information of each CRDT operation (such as the unique ID of each insert, its content, and relational pointers to other items) into a binary representation using var-int encoding and other compression techniques [19]. This results in very small payloads for typical edits.

In practical terms, a Yjs update message is a sequence of bytes that can only be fully interpreted by the Yjs library (or compatible decoders). The first byte(s) of the message identify the message type (as discussed above). The remaining bytes represent one or more operations: - For inserts, the binary data includes the operation's unique ID (comprised of the originating client identifier and a logical clock number), the length and content of the insert (which could be text, JSON, etc.), and references to the position (in terms of "left" and "right" neighbors or parent objects in Yjs's internal list structure) [20]. - For deletes, the binary data encodes a *deletion* as a range (an ID and a length) indicating which prior insert(s) are being removed [5].

This binary encoding is dense and **non-human-readable** – for example, an update might look like a short blob of bytes even if it represents a large text insertion. The Yjs team chose this format for efficiency: it ensures that applying and transmitting updates is fast and that messages are as small as possible on the wire [19]. If needed, developers can utilize helper functions like `Y.logUpdate` (for debugging) or `Y.decodeUpdate` to inspect the contents of an update in a more readable form [21]. But generally, the "compact binary chunks" are an implementation detail; they are essentially the binary-serialized CRDT diff. In summary, **Yjs updates are binary diffs containing CRDT operations**, enabling efficient synchronization. They are not in a standard format like JSON, but they encapsulate the inserts/deletes in the Yjs CRDT structure in a compressed binary form [18].

3. Yjs Snapshot (`.ysnap`) Format and Usage

Yjs provides a concept of *snapshots* (often saved with a `.ysnap` extension) which are a special kind of binary data representing a point-in-time state of a document. However, a Yjs **snapshot is not a full copy of the document's content**; it is more like a bookmark or *pointer* into the document's history. In Yjs terms, a snapshot consists of metadata (specifically, a set of client clocks and delete markers) that identifies a particular version of the document [22].

Format: A snapshot can be obtained by calling `Y.snapshot(ydoc)` to capture the current state, and then `Y.encodeSnapshot(...)` to serialize it into a binary format. The snapshot binary includes: - A state vector (clocks for each client) representing the version at the moment of the snapshot. - A set of tombstone/deletion identifiers marking which content was deleted up to that point in time [22].

Crucially, **the snapshot does *not* contain the actual document content or the full history of edits** [22]. It's a compact representation that says "this is what has been applied (and removed) up to time T," but without the actual inserted text/objects.

Usage for state reconstruction: Because a snapshot lacks the actual content, you cannot use it alone to reconstruct a past document state from scratch. To get the actual state, you need to *apply* the snapshot to a

Yjs document that has the full set of edits. In practice, Yjs offers `Y.createDocFromSnapshot(originalDoc, snapshot)` which takes an existing Y.Doc (with all history) and a snapshot, and produces a new Y.Doc representing the document at the snapshot's point in time [23]. Essentially, the snapshot acts as a filter or lens over the full history: it tells Yjs which operations to consider as "up to that version" and which to ignore (those after the snapshot or content that was deleted by that point).

Therefore: - If you have a Yjs document with its entire edit log, you **can** use a snapshot to get a prior state (read-only). The snapshot guides the creation of a view of the document at that earlier state [24]. - If you only have the `.ysnap` file *by itself*, it's not sufficient to recover the document's content at that time – you'd need the original document's updates as well. The snapshot is akin to a commit identifier in version control, while the actual content changes reside in the history of edits.

Also note that applying a snapshot does not "rewind" the original document; it produces a separate doc (or you could reload from persistent storage and then apply). Yjs snapshots are primarily meant for implementing **history, time-travel, or read-only views** of past states, rather than as standalone persisted states. If the goal is to save the current state permanently, one would typically use `Y.encodeStateAsUpdate(ydoc)` to get a full update representing the whole document content (all edits) at present [25]. That full update can be stored or later applied to an empty doc to reconstruct the latest state. In contrast, `.ysnap` files are for versioning and need the context of the full doc to be meaningful.

In summary: The `.ysnap` binary snapshot contains a set of clocks/IDs marking a version, and **by itself represents only the current state (structure) at that version, not the content**. It cannot reconstruct earlier states without the full history; it must be combined with the document's edit log to materialize a past state [22]. It is a tool for accessing previous states (when used with the original data), not an independent full-state backup.

4. The Nature of the Yjs `clock` Variable (Logical vs. Wall Time)

In Yjs (and most CRDTs), the term *clock* does **not** refer to wall-clock time. Instead, it is a **logical clock** – essentially a per-client sequence number used to order operations in the CRDT. Every time a client produces a new operation (an insert or delete), that operation is assigned an ID consisting of the client's unique identifier and the next clock value (which increments from the last value that client used) [26] [27]. This clock is often referred to as the "Lamport timestamp" or sequence for that client.

Concretely, if one client (say client ID 42) has made 10 edits so far, its next insert will have an ID like `{ client: 42, clock: 10 }` (if clocks start at 0) or 11 (if starting at 1). Another client will have its own independent clock for its edits. The Yjs algorithm uses these tuples to establish a **total order** of operations without any central time source. The ordering is achieved by first sorting by the client ID (which is unique per document session) and then by the clock number, along with additional structuring rules to intermix concurrent inserts in a sensible way.

Important details about this `clock`: - It is **not related to system time or timestamps**. It doesn't matter what the real time is; the clock is just a counter. Thus, it doesn't involve server vs. client time discrepancies or timezone issues at all. - It functions as a **logical Lamport clock** (or more precisely, a component of a vector clock). In fact, Yjs maintains a *state vector* internally which is essentially a map of each known client's ID to the clock *value of the next expected operation* from that client [28]. This state vector serves as a summary of "how many ops from each client have I seen."

Because each client's operations are numbered in increasing order, and all operations are eventually disseminated, Yjs can use the pair `{client, clock}` to uniquely identify each operation and also detect causal ordering or missing updates. The "clock" is akin to a Lamport clock in that it provides a partial ordering (it orders events from the same client, and combined with causality rules helps order events across clients in a consistent way). Yjs's conflict-resolution strategy relies on these identifiers to interweave concurrent changes deterministically [6] [7].

To directly answer the question: - **Is `clock` wall time?** No. It has nothing to do with real time. It's not server time, not client device time, and not shared across clients. - **Is it a logical clock?** Yes. It's a strictly logical counter (per client) that forms part of Yjs's Lamport-esque timestamping. It is effectively a component of a vector clock. One can think of each client's `clock` value as that client's own Lamport clock, which increments on each new event. When Yjs syncs, it shares these values (via state vectors) to know what each side has seen [26] [28].

Thus, `clock` is used for ordering and identifying operations in the CRDT. It ensures consistency and *causal ordering* of events without reliance on physical time. This is crucial for CRDTs to work in offline or diverged scenarios: even if two edits occur "at the same time" on different machines, they will have distinct `{client, clock}` IDs, and the CRDT algorithm can use those to decide a stable order for the content.

5. Criteria for a Data Structure to be a CRDT (Theoretical Qualifiers)

A **Conflict-Free Replicated Data Type (CRDT)** is a class of data structures/algorithms that by design **guarantee eventual consistency** across distributed replicas without requiring any central coordination or locking. There are formal criteria that qualify a data structure as a CRDT, which were established in academic literature (notably by Shapiro, Preguiça, Baquero, Zawirski in 2011 [29]). In essence, for something to be considered a CRDT, it must fulfill the following:

- **Concurrent, independent updates**: The algorithm allows any replica to be updated locally at any time, even while disconnected, without immediate synchronization [30]. In other words, users can make changes on different replicas concurrently.

- **Automatic conflict resolution**: The data type's definition includes a deterministic resolution strategy for concurrent updates such that no conflicts require manual resolution [31]. All operations are designed to *merge* in a well-defined way. This can be achieved via a well-defined *merge function* (in state-based CRDTs) or by making operations commutative (in operation-based CRDTs).

- **Eventual convergence**: No matter in what order or how many times updates are applied at each replica (as long as every operation is eventually delivered to every replica), all replicas are guaranteed to end up in the **identical state** (given the same set of updates) [32] [33]. This is the *strong eventual consistency* property. Formally, for state-based (convergent) CRDTs, the merge function must form a monotonic join-semilattice: merges are *commutative*, *associative*, and *idempotent* [34]. For operation-based (commutative) CRDTs, all update operations (or their effects) must commute under concurrent execution, or the system ensures they are applied in a causally consistent way such that the end result is independent of message ordering [35].

Tests or proofs: In practice, to establish that a design is a CRDT, researchers/developers use a combination of **formal proofs and reasoning** as well as **empirical testing**: - *Formal methods*: One can prove a data type is a CRDT by showing it meets the above mathematical properties. For example, one might prove that the state merge function is a mathematical *join* operation (ensuring convergence) or that every concurrent operation commutes. Academic papers often include proofs or rely on previously proven CRDT designs. There are also formal verification approaches (using model checking or theorem proving) to verify CRDT convergence properties [36] [37]. - *Testing*: While testing alone cannot *prove* a general property, developers can construct test scenarios where operations are applied in different orders on different replicas and then automatically compare end states. A thorough test suite might include random operation sequences, concurrent bursts, etc., to increase confidence that the implementation converges in all cases. If all such tests yield identical states across replicas, it suggests the algorithm is likely correct (though formal proof gives stronger guarantee) [38]. Notably, the community sometimes employs reference CRDT implementations to cross-check behavior or uses frameworks to simulate network reorderings.

Additionally, adhering to known CRDT patterns or frameworks is a way to assure CRDT properties. For instance, if one designs a new data type by composing known CRDTs or by following known constructs (like using tombstones for deletes, tagging operations with unique IDs and causal metadata), it can inherit the CRDT guarantees. But ultimately, the litmus test is **convergence under all interleavings**: if you can demonstrate or reason that any two replicas that have applied the same set of updates (in any order) will have equivalent state, then you have a CRDT.

In summary, a data structure & algorithm qualifies as a CRDT if it allows uncoordinated concurrent updates and **mathematically guarantees convergence** without external conflict resolution. This typically means designing the merges or operations to be commutative and idempotent. One can establish something as a CRDT by providing a proof of these properties (as done in the original CRDT literature) or by extensive testing for the absence of divergence. The hallmark of a CRDT is that *all replicas reconcile to the same state automatically* [31] [34]. If an algorithm meets that requirement (and the updates are applied in a causal or eventual-delivery network), it can be deemed a CRDT.

6. Role of the Yjs Server in Collaboration (Dumb Relay vs. Smart Processor)

Yjs is designed such that the heavy lifting of conflict resolution is done on the client side (within the Yjs library), not on the server. This is a deliberate contrast to Operational Transform (OT) systems. In a typical Yjs deployment using the provided WebSocket server (e.g. **y-websocket** or similar), the server's responsibilities are minimal. **The Yjs server primarily acts as a relay and a state store**, with very little logic of its own.

Key functions of the Yjs server:

- **Broadcasting document updates:** When a client sends a document update (a `[0,2]` message containing an edit), the server will accept that binary update and then forward it to all other connected clients of the same document/room. The server does not need to interpret the content of the update beyond perhaps applying it to its own copy of the document state. It **does not transform or reorder operations** – since Yjs updates are designed to commute, the server just disseminates them. This means the server can be relatively "dumb." As one source notes, the server rarely needs to do any compute-intensive work; it can simply store and forward edits, making the system very scalable [39] .

- **Initial sync (state exchange):** When a client first connects, an initial synchronization occurs. Typically, the client will send a SyncStep1 message `[0,0]` with its state vector. The server, which keeps an in-memory copy of the document (or has it in a database), will compute which edits the client is missing and reply with those (a SyncStep2 `[0,1]` message) [11] . Similarly, the server might request the client's missing edits if the server is behind (though in a single server setup, usually the server is the source of truth for stored edits). In essence, the server does use the Yjs library to *apply updates and generate diff updates*, but this is straightforward: it merges the new updates into its Y.Doc and uses Yjs's `encodeStateAsUpdate` or `diff` functions to produce the reply. It's not performing application-level conflict resolution – that logic is built into Yjs's data structure.

- **Awareness relay:** The server also relays awareness (presence) messages (type `1`). When a client broadcasts an awareness update (e.g., "user X is online with cursor at position 5"), the server forwards that to the other clients. The server might track which clients are currently in the room and possibly their last awareness state, but it doesn't interpret it beyond forwarding. The *offline timeout* for awareness is typically handled on clients (each client drops peers after 30s of no update) [15] , not by the server forcibly. Some server implementations might send periodic pings or use the awareness updates themselves as keep-alive signals (since any awareness update from a client will get broadcast, acting as a heartbeat).

- **Persistence (optional):** The base y-websocket server keeps the document in memory as long as clients are connected, and it can optionally persist data (some implementations integrate with databases or use the y-redis provider for scaling). When the last client disconnects, the server could drop the document from memory unless configured to save it. If a new client later connects, the server might have a snapshot or update log stored which it will use to sync the client. Even in persistence, the server is mostly storing the Yjs update data (the CRDT state vector and updates). It isn't doing something like line-by-line diffing or conflict resolution itself; it relies on Yjs's data model to handle those aspects.

Does the server originate any messages on its own? Generally, **no new document operations are generated by the server** in the default model. The server doesn't act as an editor; it doesn't, for example, generate a "[0,2] update" unless it is replying with stored updates during sync. One exception could be if a server has a feature to inject operations (say, a server-side bot or automation that updates the doc), but that would be an application-specific extension where the server would internally use Yjs to insert content and then broadcast it. By default, the server's only autonomous messages are the SyncStep2 replies triggered by a client's SyncStep1 request, and possibly periodic awareness broadcasts if implemented (some setups use the client to periodically send awareness, others might have server broadcast a null awareness to flush out inactive clients – but in Yjs's reference implementation, clients handle it).

So, the Yjs server is mostly a **hub that echoes messages**: - It receives updates from one client and broadcasts to others [40] . - It does a one-time calculation of diffs during initial sync (full state exchange) using the Yjs library's helper functions [40] . - It manages connections/rooms and maybe basic auth, but *not the operational transformation of content*.

This "dumb server, smart client" approach is a fundamental advantage of CRDT-based systems like Yjs. As noted in the literature, because the CRDT algorithm guarantees convergence, the server does not need to intervene or reorder operations, unlike an OT server which must detect and correct conflicts. Thus, the server can be stateless or thin. In fact, you could have a pure peer-to-peer model and skip the server entirely (Yjs supports WebRTC or other provider modules for P2P); the server in client-server mode is just simplifying topology but not acting as an arbiter of edits.

To quote a summary: *"The server can be 'dumb', and just store and broadcast edits. It does not need to process each incoming edit...which makes it very scalable"* [39] . All the heavy logic (like integrating an edit into the text, handling concurrent inserts at the same position, etc.) is done by Yjs on the client and within the CRDT itself. The server's correctness does not depend on understanding document semantics — it just needs to reliably deliver messages. In Yjs's default implementation, once it broadcasts a client's update to others, those clients apply it and the document state converges. The server doesn't correct or adjust anything in those messages.

In conclusion, **the Yjs server primarily rebroadcasts and persists updates**. It *does not transform or reorder operations*, and it doesn't initiate new edits on its own (besides responding to sync requests with stored data). Its tasks are managing connections, perhaps storing the combined document state, and ensuring every client gets every other's updates. This simple server role is a key reason Yjs can scale well and be integrated easily — the server can even be a simple script or use provided packages like `@y-websocket-server` without deep application logic. All the CRDT intelligence lives in the Yjs client/library, which is exactly where conflicts are resolved and the final state is computed [39] [40] .

Sources:

1. Dovetail Engineering – *Yjs Fundamentals, Part 1: Theory* [2] [3]
2. Dovetail Engineering – *Yjs Fundamentals, Part 2: Sync & Awareness* [41] [39]
3. Yjs Documentation – *Internals and Updates* [28] [42]
4. Yjs Community Discussions (Raine's explanations) [11] [22]
5. Wikipedia – *Conflict-Free Replicated Data Type* (formal definition and properties) [31] [34]

---

[1] [2] [3] [4] [5] [20] Yjs Fundamentals - Part 1: Theory | by Dovetail Engineering | Dovetail Engineering | Medium
https://medium.com/dovetail-engineering/yjs-fundamentals-part-1-theory-232a450dad7b

[6] [7] Easy to understand explanation of how Yjs works - Yjs Community
https://discuss.yjs.dev/t/easy-to-understand-explanation-of-how-yjs-works/2835

[8] How to decode the y-websocket message - Yjs Community
https://discuss.yjs.dev/t/how-to-decode-the-y-websocket-message/2203

[9] [10] [11] [14] [17] "read-only" or one-way only sync - Yjs Community
https://discuss.yjs.dev/t/read-only-or-one-way-only-sync/135

12 13 Resync Interval - Unsaved changes - Yjs Community
https://discuss.yjs.dev/t/resync-interval-unsaved-changes/1053

15 16 Awareness | Yjs Docs
https://docs.yjs.dev/api/about-awareness

18 21 28 42 Document Updates | Yjs Docs
https://docs.yjs.dev/api/document-updates

19 Kevin's Blog
https://blog.kevinjahns.de/are-crdts-suitable-for-shared-editing

22 23 24 25 How to decode the snapshot of yjs - Yjs Community
https://discuss.yjs.dev/t/how-to-decode-the-snapshot-of-yjs/2299

26 27 39 40 41 Yjs Fundamentals — Part 2: Sync & Awareness | by Dovetail Engineering | Dovetail Engineering | Medium
https://medium.com/dovetail-engineering/yjs-fundamentals-part-2-sync-awareness-73b8fabc2233

29 30 31 32 33 34 35 Conflict-free replicated data type - Wikipedia
https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

36 [PDF] Type-Checking CRDT Convergence - Programming Group
https://programming-group.com/assets/pdf/papers/2023_Type-Checking-CRDT-Convergence.pdf

37 [PDF] Modular Verification of State-Based CRDTs in Separation Logic
https://iris-project.org/pdfs/2023-ecoop-crdts.pdf

38 CRDTs go brrr - Seph
https://josephg.com/blog/crdts-go-brrr/