Authenticated Transfer
Internet-Draft
Intended status: Standards Track
Expires: 6 December 2026

D. Holmgren
B. Newbold
Bluesky Social

4 June 2026

Authenticated Transfer: Synchronization
draft-holmgren-at-synchronization-00

Abstract

This document describes synchronization mechanisms for public repositories as part of the Authenticated Transfer Protocol (ATP). It specifies both a low-latency streaming protocol over WebSocket, and a full-repository fetch mechanism over HTTP.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-holmgren-at-synchronization/.

Discussion of this document takes place on the Authenticated Transfer Working Group mailing list (mailto:atp@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/atp/. Subscribe at https://www.ietf.org/mailman/listinfo/atp/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-atp/drafts.

Status of This Memo

Table of Contents

1. Introduction

The Authenticated Transfer Protocol (ATP) enables the creation of decentralized networks for publication of self-certifying data. An introduction to the overall protocol architecture is given in [AT-ARCH].

The protocol provides efficient synchronization mechanisms for propagating public repository state changes across the network, supporting both low-latency streaming updates and bulk synchronization scenarios. Synchronization can take place between any two parties, from an upstream publisher to a downstream consumer. Intermediate parties can redistribute ("relay") data, and consumers can cryptographically verify the integrity and authenticity of received repository data. This allows for flexibility in network topology to improve overall network resilience and efficiency.

Repositories are complete (they contain all current public records for the account), and consumers can confirm the integrity over the entire repository to detect dropped or withheld updates. The protocol allows consumers to maintain partial replicas (eg, of only specific record types). It is also possible to request verifiable "inclusion proofs" for individual records on demand.

This document describes two synchronization mechanisms. Complete repositories can be serialized (as described in [ATREPO]) and fetched over HTTP as a snapshot. Updates to one or more repositories can be distributed as a stream of messages. This document describes both a generic structure and semantics for streaming messages, and a specific WebSocket transport and message encoding scheme.

2. Account Hosting

Each node in the network which synchronizes, stores, and distributes repository data maintains hosting status for each account. The status indicates whether the account is overall "active", or has been temporarily or permanently removed from the network, in which case repository data should not be synchronized further. Hosting status can be set by the account holder themselves or their canonical hosting provider, and changes propagate to downstream consumers throughout the network. Each downstream service or node in the network may set a local inactive hosting status, declining to distribute that account's repository data.

Each account has a persistent identifier which can be resolved to both a public key and a canonical hosting location. Account identifier systems and their resolution mechanisms are out of scope for this document. Account hosting status is maintained and transmitted over the synchronization protocol, separate from the lifecycle of account identifiers.

The hosting status itself for accounts may always be redistributed, even for inactive accounts.

2.1. Hosting Status

Account hosting status at any point in time can be summarized as the boolean state of being "active" or not. If the account hosting status is not active, repository data for that account MUST NOT be redistributed. Additional context may be provided using a "status" vocabulary, represented as a string. Account status is represented and transmitted as an active boolean and a status string together.

The defined status values and their meanings are:

* deleted (active is false): the user or host has deleted the account. Account data SHOULD be removed from the service's infrastructure within a reasonable time frame. Implied to be permanent, but MAY be reverted.

* deactivated (active is false): the user has temporarily paused the account. Account data MUST NOT be redistributed but does not need to be deleted from infrastructure. Implied time-limited.

* takendown (active is false): the host or service has taken down the account. Implied to be indefinite in duration, but MAY be reverted.

* suspended (active is false): the host or service has temporarily paused the account. Implied time-limited.

Two additional status values are relevant to the synchronization process itself, and do not imply that the overall hosting status is inactive:

* desynchronized (active MAY be true): the service has detected a problem synchronizing the account's repository and may be missing content.

* throttled (active MAY be true): the service has paused processing of new content for this account because a rate limit has been exceeded.

New status values may be defined in the future. Producers MAY emit status strings not listed above, and consumers MUST tolerate unrecognized values. Consumers MUST use the active boolean as the authoritative indicator of overall account visibility, treating the status string as clarification that may inform more specific behavior (for example, whether to delete cached data versus retain it pending reactivation).

Producers expose an HTTPS request-response operation that, given an account identifier, returns the producer's current hosting status for that account. This allows consumers to query the present state of an account without subscribing to the message stream — for example, when establishing initial state for an account they have not seen before, or when reconciling diverging upstream reports.

The details of the HTTPS request endpoint, the URL path, and the response media type are not specified by this document.

2.2. Status Propagation

Account hosting status is not cryptographically authenticated. Status propagates hop-by-hop via streaming synchronization: each node emits an #account message (Section 4.4.4) to downstream consumers when the hosting status for an account changes. For intermediate synchronization nodes, this includes changes driven by an #account message received from an upstream.

Intermediaries MAY override their upstream's status. For example, a relaying node may take down an account that an upstream still reports as active. Such overrides are propagated downstream as #account messages from the intermediary.

When an upstream service is unreachable, downstream services SHOULD retain the previously reported status for some implementation-defined period rather than immediately changing the account to an inactive state. This preserves availability across short upstream outages, and increases resiliency of the network.

When account status reported by different upstreams diverges (for example, due to differing moderation policies, or a transient network partition between an upstream and its own upstream), services apply their own policies to reconcile. Querying the account's current authoritative hosting service directly is one way to resolve such ambiguity.

3. Full Repository Sync

Consumers can retrieve a full serialized snapshot of an account's current repository at any point in time. This can be used to initialize synchronization state for the account during a bootstrap or backfill phase, or to re-synchronize (Section 4.6) and reconcile after any discontinuity in the streaming synchronization mechanism. It is also an option for applications and use-cases which do not require continuous updates or synchronization over time.

Producers expose an HTTP API endpoint which takes an account identifier as a parameter, and returns the serialized repository as the response body. If the account hosting status at the producer is not "active", this is indicated in an error response.

A producer MAY redirect the request to another producer that holds the requested data — for example, a relaying service redirecting to the account's canonical host, or to a mirroring service with the content cached. Consumers SHOULD follow such HTTP redirects when re-synchronizing per Section 4.6.

4. Streaming Sync

This section describes a low-latency streaming synchronization mechanism which allows consumers to receive repository updates with minimal latency through a pull-based WebSocket [RFC6455] connection. Streams can contain updates from many distinct account repositories, even aggregating all updates in the network. In addition to repository updates, they include updates to account hosting status and network identities. Whether encompassing the full network or any subset of it, a stream is often referred to as a "firehose".

To ensure reliable delivery, each message on a given stream is given a monotonically-increasing sequence number. Consumers can track their progress on processing messages and reconnect to the stream using the last-processed sequence number as a cursor value as needed. Producers can maintain a "backfill window" of recently transmitted messages. All consumers receive the same messages in the same order with the same sequence numbers.

The stream synchronization mechanism allows consumers to maintain complete indices of authenticated repository contents without needing to store complete copies of the repository MST structure. This significantly reduces storage overhead.

4.1. Repository Diffs

A repository diff carries the data that changed between two repository revisions: the new commit, any new MST nodes, and any created or updated record blocks. Applying a diff to a copy of the prior repository state results in the complete repository at the new revision. Repository diffs are used in the streaming synchronization mechanism.

The commits within diffs are signed. Receiving parties can always verify those signatures, and the integrity of the blocks in the diff itself. But unless the receiving party has a full copy of the repository just prior to the diff, it can not verify the overall integrity of the diff or the final state of the repository. In particular, if record deletion operations were included in the diff, the receiving party can not enumerate or verify which records were impacted just from the diff.

This section describes an "operation inversion" mechanism which allows receiving parties to verify the integrity of diffs when combined with metadata about the record-level operations encapsulated by the diff.

4.1.1. Diff Serialization Format

Diffs use the same serialization format as complete repositories (described in [ATREPO]), with the commit block serving as the root. A diff MUST include:

* The new commit block.
* All created and updated record blocks.
* All MST nodes in the current repository that did not exist in the prior revision.

Required blocks MUST be included in the diff regardless of their presence in earlier repository history. For example, if an MST node was previously present in the repository, then deleted, and subsequently reintroduced during the range that the diff represents, the diff MUST include that block.

Deleted records and prior versions of updated records are excluded from diffs.

With the exception of deleted record data, a diff MAY include additional blocks; receivers SHOULD ignore them.

4.1.2. Operation Inversion

A diff can be accompanied by an explicit operation list declaring the record-level creates, updates, and deletes it represents, along with a claimed previous repository tree root hash. Operation inversion verifies that this declared list is accurate and complete.

To invert a diff against its declared operations:

1. Extract the diff's MST nodes into a partial tree structure
2. For each entry in the operation list, apply the inverse operation to the partial MST: each "create" becomes a "delete", "delete" becomes "create", and "update" reverts the record value to the previous version
3. Compute the root hash of the resulting MST
4. Compare the computed root hash to the claimed previous root hash

If the hashes match, the operation list is accurate and exhaustive. If they differ, either the operation list is incomplete or the diff is internally inconsistent; in either case the diff MUST be rejected.

Producers of diffs intended to support operation inversion MUST include, in addition to the blocks required by Section 4.1.1, the MST nodes for keys directly adjacent (in lexicographic order) to mutated keys. Without these adjacent nodes, the inverse operation cannot be correctly applied to the partial MST. Diffs carried in streaming synchronization messages (Section 4.4.2) MUST satisfy this requirement, since stateless consumers rely on operation inversion for verification.

Receivers can track repository root hash values for each account and verify the claimed root hashes provided with the diff. This fixed-size state is significantly smaller than maintaining full repository data.

4.2. WebSocket Transport

A consumer (client) establishes a WebSocket connection [RFC6455] to the producer's (server) stream endpoint. Secure WebSockets (using TLS) MUST be used for any internet-facing deployment.

Once the connection is established, the producer sends a sequence of binary WebSocket frames to the consumer. Each frame carries a single message. Servers SHOULD ignore stream messages sent by the consumer: the protocol defined in this document is server-to-client only.

Servers and clients SHOULD implement a keepalive system using Ping and Pong WebSocket messages.

4.2.1. Frame Format

Each binary WebSocket frame contains two CBOR-encoded objects concatenated together: a header followed by a payload. Both objects MUST follow the deterministic CBOR encoding rules defined in [ATREPO].

The header contains:

* op (integer, REQUIRED): the frame operation. The value 1 indicates a normal message; the value -1 indicates an error.

* t (string, REQUIRED when op = 1): the message-type name, prefixed with #. For example, #commit for a commit message.

The payload is a CBOR object whose schema is determined by the message type indicated in the header. Payloads are always CBOR objects, never arrays or scalars.

Producers MAY include additional fields beyond those defined for a given message type, and consumers MUST tolerate unknown fields. Strict schema validation is not performed on stream messages.

A frame MUST NOT exceed 5 MB in total size, inclusive of the header, payload, and CBOR encoding overhead.

4.2.2. Error Frames

When op is -1, the frame is an error frame. The payload contains:

* error (string, REQUIRED): a short machine-readable error name.

* message (string, OPTIONAL): a human-readable description of the error.

After sending an error frame, the producer MUST close the WebSocket connection.

4.2.3. Pre-Upgrade HTTP Errors

If the producer rejects the WebSocket upgrade request itself, it responds with a standard HTTP status code rather than an error frame. Response bodies SHOULD be JSON containing error and message fields matching the error-frame schema, but consumers MUST tolerate other body content.

4.3. Cursors and Resumption

Synchronization streams include per-message sequence numbers to improve transmission reliability. Sequence numbers are positive integers that increase monotonically across the stream. Sequence semantics are flexible, and they may contain arbitrary gaps between consecutive messages.

Consumers track the last sequence number they successfully processed and can specify this as a cursor when reconnecting to receive any missed messages within the provider's backfill window. Consumers are responsible for managing and persisting cursor state themselves: producers do not maintain consumer-specific state across connections. The scope of a cursor is the (hostname, endpoint) pair: a cursor value is meaningful only when reconnecting to the same host and stream endpoint that issued it.

Sequence numbers MUST NOT be repeated by producers on the same (hostname, endpoint) pair. If a producer must reset sequence numbers for any reason, it MUST start with a number higher than any previously broadcast.

Sequence numbers are integers in the range [1, 2^53). The upper bound is chosen so that cursors are exactly representable in 64-bit IEEE-754 floating point.

Stream behavior depends on the cursor value specified during connection:

* *No cursor specified*: The provider begins transmitting from the current stream position, providing only new messages generated after the connection is established.

* *Future cursor*: When the requested cursor exceeds the current stream sequence number, the provider sends an error message and closes the connection.

* *Cursor within backfill window*: The provider transmits all persisted messages with sequence numbers greater than or equal to the requested cursor, in order, then continues with the stream once caught up.

* *Cursor older than backfill window*: The provider sends an informational message indicating that the requested cursor is too old, then begins transmission at the oldest available message, sends the entire backfill window, and continues with the stream.

* *Cursor value of 0*: The provider treats this as a request for the complete available history, starting at the oldest available message, transmitting the entire backfill window, then continuing with the stream.

4.4. Message Types

The repository synchronization stream uses four message types: #commit, #sync, #account, and #identity. This section describes the schema and semantics of these messages. Some fields are common across all message types.

4.4.1. Common Message Fields

The following fields are common to all message payloads:

* seq (integer, REQUIRED): the sequence number (cursor; see Section 4.3) of this message.

* did (string, REQUIRED): the account identifier of the repository this message concerns. For historical reasons the #commit message uses the field name repo rather than did for this purpose; the value has the same meaning.

* time (string, REQUIRED): an ISO 8601 datetime string indicating when the message was emitted. This timestamp is informational and is not authoritative for any verification purpose.

4.4.2. #commit Message

A #commit message represents a repository update, as an atomic set of record operations. The message contains a repository diff combined with supporting metadata.

The payload contains:

* seq (integer, REQUIRED): see Section 4.4.1.

* repo (string, REQUIRED): the account identifier of the repository (see Section 4.4.1; this is the historical name of the did field). MUST match the did field in the commit object enclosed in blocks.

* time (string, REQUIRED): see Section 4.4.1.

* rev (string, REQUIRED): the new revision identifier of the repository after these modifications. MUST match the rev field in the commit object enclosed in blocks.

* since (string, REQUIRED, nullable): the revision identifier of the repository immediately prior to this commit. May be null only for the first commit of a repository.

* commit (hash link, REQUIRED): reference to the new commit object. MUST match the hash of the commit object enclosed in blocks.

* blocks (byte string, REQUIRED): the serialized diff (as defined in Section 4.1) carrying all blocks required to invert and verify the operations in this message.

* ops (array, REQUIRED): the set of record operations encapsulated by this message. Multiple operations on the same record (path) are not allowed within a commit. Each entry is an object containing:

  - action (string, REQUIRED): one of create, update, or delete.

  - path (string, REQUIRED): the repository path of the record being mutated.

  - cid (hash link, REQUIRED, nullable): the hash link of the new record at this path, or null for delete actions.

  - prev (hash link, OPTIONAL): the hash link of the prior record at this path. Present for update and delete actions; absent for create.

* prevData (hash link, REQUIRED): the root hash of the repository's MST in the previous revision (the data field in the commit object). Used for operation-inversion validation as described in Section 4.5.

* tooBig (boolean, REQUIRED): retained for compatibility with earlier versions of this protocol. Producers MUST emit this field with the value false. Consumers MUST ignore the field's value.

* blobs (array, REQUIRED): retained for compatibility with earlier versions of this protocol. Producers MUST emit this field as an empty array. Consumers MUST ignore the field's contents.

A #commit message MUST contain no more than 200 entries in ops. The blocks field MUST NOT exceed 2 MB. Any single record block within blocks MUST NOT exceed 1 MB. Repository updates exceeding these limits MUST be communicated through #sync message instead.

A #commit message with an empty ops array (e.g., a commit issued solely to advance rev after a key rotation) is valid.

Note that the full message is _not_ cryptographically authenticated end-to-end (from the origin account itself). Only the commit object contained within the blocks field is signed, with other blocks covered by proof chains. The ops array is _not_ authenticated and must be verified via operation inversion.

4.4.3. #sync Message

A #sync message declares the current state of a repository, regardless of the previous state. Sync messages are emitted when commit-message continuity cannot be maintained: large mutations exceeding the limits in Section 4.4.2, recovery from data loss or corruption, or account migration between hosting providers.

The payload contains:

* seq (integer, REQUIRED): see Section 4.4.1.

* did (string, REQUIRED): see Section 4.4.1. MUST match the did field in the commit object encapsulated in blocks.

* time (string, REQUIRED): see Section 4.4.1.

* rev (string, REQUIRED): the current revision identifier of the repository. MUST match the rev field in the commit object encapsulated in blocks.

* blocks (byte string, REQUIRED): a serialized stream containing only the current commit object. Receivers reconstruct full repository state by fetching the complete repository as described in Section 4.6.

A #sync message provides a reset point that signals consumers to resynchronize against the current authoritative state without requiring knowledge of the intervening changes.

A #sync message with a rev which is lower or equal to the previously tracked revision for an account would constitute a "rollback" and should be ignored. The commit object signature (within the blocks field) should also be verified, and the message rejected if validation fails.

4.4.4. #account Messages

An #account message indicates a change in account hosting status for an indicated account. See Section 2.1 for details and semantics.

The payload contains:

* seq (integer, REQUIRED): see Section 4.4.1.

* did (string, REQUIRED): see Section 4.4.1.

* time (string, REQUIRED): see Section 4.4.1.

* active (boolean, REQUIRED): whether the account is currently active on the emitting service.

* status (string, OPTIONAL): a short status code describing the account state.

Note that account messages are _not_ cryptographically authenticated end-to-end, and that they have hop-by-hop semantics.

4.4.5. #identity Message

An #identity message indicates a possible change to the resolution result of an account identifier.

Note that identity messages are _not_ cryptographically authenticated end-to-end. Consumers SHOULD invalidate any cached identity metadata for the named account on receipt of this message, and then re-resolve the identifier.

The payload contains:

* seq (integer, REQUIRED): see Section 4.4.1.

* did (string, REQUIRED): see Section 4.4.1.

* time (string, REQUIRED): see Section 4.4.1.

* handle (string, OPTIONAL): included for historical reasons. Non-authoritative.

#identity messages are best-effort: producers MAY emit them redundantly when no underlying change has occurred, and MAY fail to emit them when a change has occurred. Consumers SHOULD NOT rely on #identity messages as the sole signal of identity change.

4.5. Commit Validation

Validating a #commit message establishes both that the message is internally consistent (its declared operations match the diff it carries) and that it matches the prior state of the account’s repository from the perspective of the consumer.

For each #commit message received, consumers MUST perform the following steps:

1. Verify wire-level fields: that the frame parses as deterministic CBOR, that the payload satisfies the schema in Section 4.4.2, and that the size limits in Section 4.4.2 are not exceeded. Otherwise the message MUST be rejected.

2. Parse the repository diff from blocks, verifying the deterministic CBOR encoding, schema, and syntax of all fields of the commit and MST nodes. All created and updated record blocks must be present, but the content and internal structure of records are not validated at this stage. If the repository diff is invalid, the message MUST be rejected.

3. Apply operation inversion to the repository diff MST using the ops array, and match against the message's claimed prevData, as per Section 4.1.2. If inversion fails, the message MUST be rejected.

4. Verify the commit signature using the signing key resolved from the account identifier. If the signature is invalid, the message MUST be rejected.

5. Confirm that the message's rev is strictly greater than the previously observed rev for this account. Otherwise the message MUST be ignored.

6. Cross-check the message's prevData field against the consumer's previously seen data for this account. If they differ, the consumer has become desynchronized for this account and MUST initiate re-synchronization as defined in Section 4.6.

A signature failure at step 4 might indicate a recent key rotation rather than a malicious commit. Consumers SHOULD refresh the cached identity for the account and retry verification before rejecting the message.

4.6. Re-synchronization

When a consumer detects desynchronization, either through a disjunction in commit history or a sync message that does not match their local state, they must perform a complete re-synchronization process to restore consistency with the current repository state.

Re-synchronization requires fetching and processing the full repository structure, though the record contents themselves are optional depending on the consumer's needs. If the repository data is delivered in pre-order traversal, it can be validated incrementally as it is received.

Parsing the repository structure produces a mapping of keys (repository paths) to record versions (hashes) that represents the complete repository state. This key-to-hash mapping can be compared against existing local state to identify discrepancies and re-establish synchronization. Once validated, this mapping establishes the new repository state against which future commit messages can be applied.

Consumers SHOULD prefer requesting full repository data from their direct upstream rather than the resolved canonical host for the repository. Direct upstreams MAY coalesce and cache snapshot requests, or redirect consumers to alternative sources where appropriate. This reduces correlated load spikes on canonical hosts caused by re-synchronization message broadcast.

During the re-synchronization process, any incoming commit messages for the repository should be buffered rather than processed immediately. Once re-synchronization completes successfully, these buffered commits can be validated and applied in sequence to bring the consumer fully up to date with the current repository state.

5. Security Considerations

Security considerations for the repository format itself, including CBOR processing limits and MST structural validation, are covered in [ATREPO].

5.1. Resource Abuse

A producer that routinely emits #sync messages could cause consumers to repeatedly fetch full repository snapshots, which is substantially more expensive than processing #commit messages. Similarly, a producer that rapidly updates the same key or issues many small commits can amplify message volume and bandwidth costs on downstream consumers. Consumers should apply rate limits and bandwidth budgets per repository, and may disconnect or deprioritize producers whose message patterns appear abusive.

5.2. Repository Rewinds

Intermediaries that relay firehose messages can present a consumer with an outdated view of a repository by replaying older commits or declining to forward newer ones. The rev field on each commit can help consumers detect this: a received commit whose rev is not greater than the most recently observed rev for that repository may indicate that the producer is serving a rewound view. In situations such as this, consumers can cross-check the latest observed rev against the canonical host for the repository.

5.3. Server-Side Request Forgery

Several aspects of synchronization involve following URLs or host endpoints derived from untrusted input: account-identifier resolution, retrieval of full repository data from a hosting service, and following redirects between hosts. Consumers MUST validate URLs derived from untrusted input before issuing requests, including any URLs reached via HTTP redirects. In particular, requests to internal-network addresses, loopback addresses, and link-local addresses MUST be rejected unless explicitly permitted by configuration.

5.4. Validation Responsibility

Intermediaries that relay messages MAY apply some validation checks (for example, signature verification or size enforcement) before relaying. Consumers MUST NOT treat upstream relaying as evidence of validity: every consumer is ultimately responsible for performing the verification rules in Section 4.5 on each message it processes.

6. IANA Considerations

This document has no IANA actions.

7. References

7.1. Normative References

[ATREPO] Holmgren, D. and B. Newbold, "Authenticated Transfer Repository", June 2026.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <https://www.rfc-editor.org/rfc/rfc6455>.

7.2. Informative References

[AT-ARCH] Newbold, B. and D. Holmgren, "Authenticated Transfer: Architecture Overview", June 2026.

Acknowledgments

TODO: acknowledge

Authors' Addresses

Daniel Holmgren
Bluesky Social
Email: daniel@blueskyweb.xyz

Bryan Newbold
Bluesky Social
Email: bryan@blueskyweb.xyz