



POSSIBILITY SPACE • FULL USE CASE

05

The community that outlives its app

Posts and replies that anyone can rebuild from public data, so no single app holds the community together.

BUILD STATE

Yours to take

COMPLEXITY



Vision

A forum for some narrow interest runs for fifteen years. A few thousand people, most of the accumulated knowledge in the field, the two or three members everybody knows will answer a hard question by Tuesday. Then whoever pays for it stops: they sell, or lose interest, or the company pivots. A notice goes up. The site goes read-only, then it goes.

The people are all still alive and still interested. What makes the loss total is that the community and the software hosting it were one object. The archive was one database, the membership list was in the same place, and the only channel anyone had to the other members was the thing that just closed. Nobody can tell four thousand people where everyone went, because telling them was a feature of the site.

The same failure also arrives without a shutdown. The software stays up and changes underneath its members: the ordering starts favouring whatever pays, the rules move, a decision lands that most of the community hates. Leaving is possible, and it costs everything again, because leaving means starting over with however few people follow.

Swarm separates the two. What people write is signed by them and stored in a network nobody operates, so it stays reachable whether or not the app that displayed it does. An app becomes one way of looking at a conversation rather than the place the conversation lives. When it closes or changes hands or makes a decision its members will not accept, somebody else can put up a different view over exactly the same posts, and nothing has to be exported, migrated, or asked for. Continuity stops depending on anyone's goodwill.

It also changes who pays for the memory. Storage is prepaid and anyone can add to it, so a community can fund its own archive directly instead of hoping that whoever owns the servers stays interested for another decade.

Architecture

A social network of this kind is a pipeline rather than a place, and it has three steps. Someone writes a post and puts it on Swarm. Someone makes others aware that it exists. Something picks it up and rebuilds a readable view of everything so far. Swarm supplies the storage for the first step and the publishing surface for the third. The second step is where the only real design choice lives, and it is the one that decides how much of this arrangement anybody has to trust.

The invariant that makes the rest reversible. A post is signed by a key its author holds and stored at an address derived from its own bytes. Authorship is checkable by anyone, the post cannot be edited in place or swapped for different content, and its Swarm reference is its identity for the rest of the pipeline. Nothing downstream can rewrite what somebody said. That is why a bad aggregator is a recoverable problem rather than a fatal one.

If the set of authors is known, none of the pipeline is needed. Swarm documents a pattern where each author writes their own feed and holds its key, an index feed lists the author feeds, and a reader follows the index to discover everyone, with runnable code for each step. There is no broadcast and no indexer, because there is nothing to discover that was not already listed. An aggregator here picks what to list and cannot alter or remove what an author wrote, so delisting is visible and survivable: the author's feed keeps working for anyone holding its address. This is the right starting point for a publication with named contributors, and the wrong one for a network anybody can join. Everything below is about the second case.

Step one: write the post. The post is signed and uploaded, and the reference that comes back identifies it everywhere else. A postage stamp pays for the write, and because anyone can top up any batch, the cost of keeping an archive of posts reachable can be shared by the people who want it kept rather than carried by whoever ran the software.

Profiles come free from the same primitive, with no registry anywhere. A Single Owner Chunk address is determined by its owner and an identifier, so if the owner is the user, the address of that user's profile is something any client can compute rather than look up. A person publishes a display name, a bio and a picture reference as JSON to a feed they own, and other clients simply try the address that person's profile would be at. If something is there, it renders; if not, the person has no profile yet. There is no directory to be captured, no namespace to be squatted, and no service that can decide a profile has stopped existing.

Step two: broadcast, and this is the choice. A post sitting on Swarm is unreachable until somebody knows its reference. There are two ways to say so.

With GSOC or PSS, which means a server node. The poster sends the reference as a Swarm message, and a designated node runs a sidecar application holding a WebSocket subscription that receives it, validates what arrived, and rebuilds the readable view. GSOC has many writers mine their way into that node's neighbourhood so they all write to one address it is listening on; PSS sends a targeted encrypted message into a neighbourhood, needing the recipient's public key and at least a two-byte prefix of their address. The listener has to be a full node either way, because light and ultra-light nodes cannot subscribe to these messages at all, and for GSOC the batch paying for it has to be mutable, since each update consumes a slot. The honest limit is what that node's absence costs: posts already published stay readable, because they are on Swarm and the last rebuilt view is already written, but nothing new gets seen until it comes back. One implementation puts it exactly right about itself, and the second half of the sentence matters as much as the first: the node "is effectively the server, but it operates entirely within the Ethereum Swarm network — there is no traditional backend outside of it."

With an on-chain event, which means no server node. The poster calls a contract whose entire body is one event and passes the post's 32-byte Swarm reference. No content goes on chain, and no privileged listener exists. Ordering falls out of block number and then position within the block, so readers who have never communicated derive the same sequence. The benefit that matters is not the ordering though: it is that anyone can run the same indexing code over the same public log and check the result locally. Nobody can be stopped from appending, and nobody can remove an entry.

Step three: index and publish. Both paths converge here, and the difference between them is where this function sits rather than whether it exists. In the first, the rebuild is welded to the privileged listener and there is one of it. In the second, it stands alone and there can be any number, all deriving the same state from the same inputs. Either way the work is the same: fetch each referenced post, verify its signature, replay in order, and publish the result as a static site or as a public database written to a Swarm feed that frontends read and poll. That reproducibility is the mechanical form of the claim in the title. The aggregator holds nothing of its own, so it is rebuildable, and therefore replaceable.

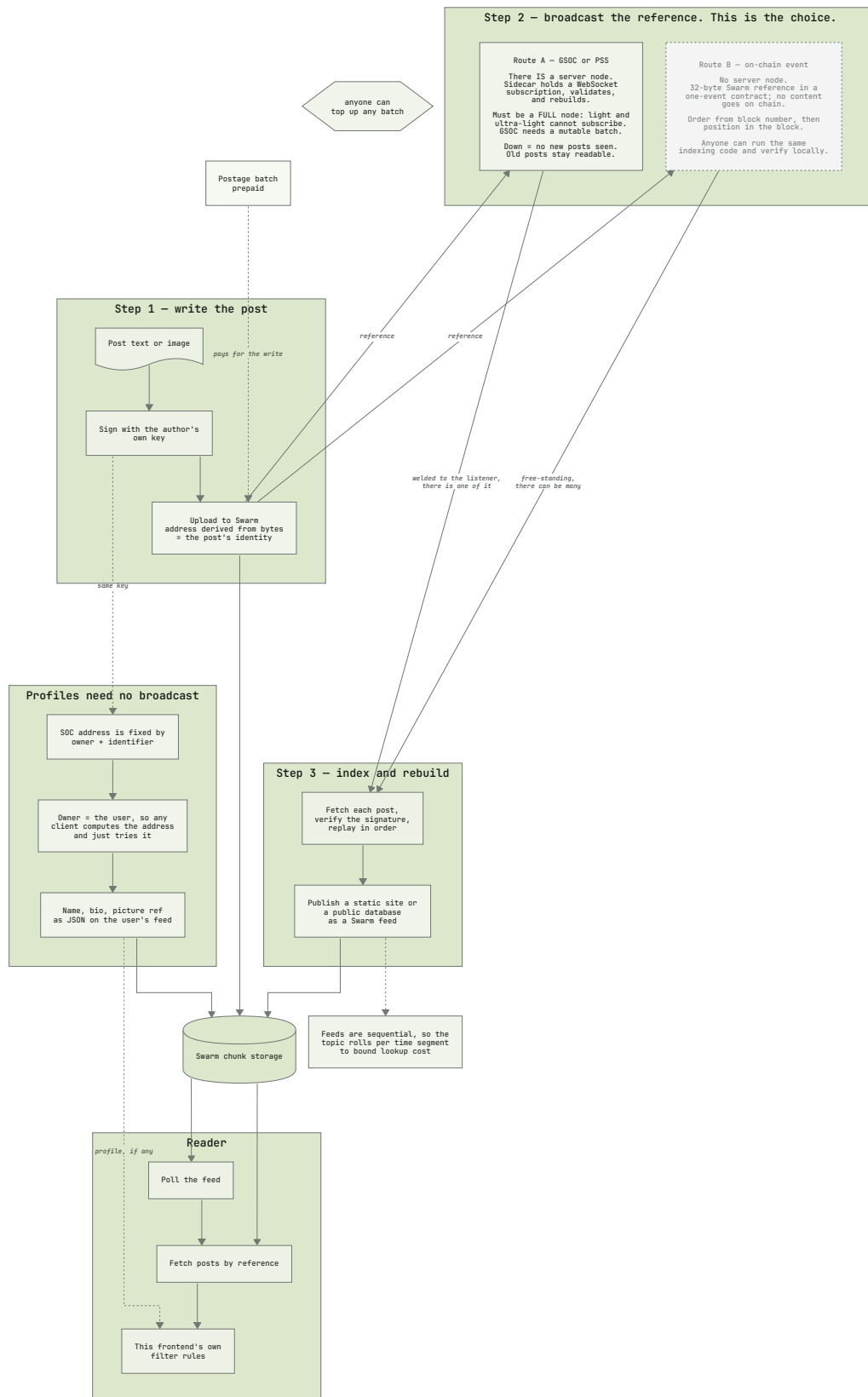
Griefing and moderation. An append point anybody can use gets abused, and these designs assume it rather than discovering it: entries that fail validation are skipped and the replay carries on. The consequence is a real trade-off and worth stating as one. Filtering becomes a property of whichever frontend a reader chooses, several frontends can apply different rules to the same material,

and none of them can make a post unavailable to somebody using a different one. Removal, in the sense a platform means the word, does not exist here.

One practical detail every implementation hits. Swarm feeds are sequential, so a reader walks the update chain to find the newest entry and lookup time grows with its length. An indexer publishing continuously therefore derives its feed topic from a time segment and starts a fresh feed when the segment rolls over, which bounds lookup to the updates inside one segment. Readers derive the same topic from the same clock, so nothing has to be announced, and republishing under the new topic as soon as the segment changes keeps a reader from finding nothing during the handover.

PRIMITIVES Content addressing Single Owner Chunks Feeds GSOC PSS Postage stamps

EXTERNAL DEPENDENCIES Gnosis Chain Solidity Foundry viem / ethers



Data flow, in the brief's three steps. A signed post is uploaded to Swarm, where its content-derived address becomes its identity. The poster then broadcasts that reference one of two ways: as a GSOC or PSS message to a full node whose sidecar listens on a WebSocket and rebuilds, or as a one-event chain call that any number of independent indexers replay. Both converge on a rebuilt view published to feeds that readers poll. Profiles need neither route, since a Single Owner Chunk address can be computed from the author's own address.

Development

Pick the broadcast mechanism first, because it decides what you have to operate and almost nothing else changes. Feeds referencing feeds needs no broadcast at all and suits a known set of authors. GSOC or PSS means running a full node with a sidecar that has to stay up for new posts to appear. A chain event means an external dependency and no privileged component, so any number of people can index and verify independently.

The write path, and the convention to get right first. A post is a signed object, and every reader has to recover the author's address from the bytes it fetched, so the serialization has to be exact. Two conventions already exist to copy. One is byte-oriented: lay out signature, author address, previous-tip reference and payload at fixed offsets and sign everything after the signature. The other is JSON-oriented: serialize with keys in alphabetical order and the signature field omitted, hash that, sign it, and put the signature back. Either works. Mixing them, or leaving serialization order unspecified, produces posts that some readers verify and others reject. Keep the user's secp256k1 key in browser storage and use its Ethereum address as the identifier.

Profiles need no infrastructure. Derive the Single Owner Chunk address from the user's own address and a fixed identifier for "profile", write the metadata there as JSON, and have clients attempt that address for any author they encounter. Treat absence as "no profile" rather than an error, cache what you find, and remember the user can update it at any time because they own the feed.

If you broadcast with GSOC or PSS. Mine a writer key against the collecting node's overlay address so chunks land in its neighbourhood, and give it to every client. On the node, hold one WebSocket subscription and validate on arrival: recover the signature to the claimed author, require the claimed previous entry to be the current tip so ordering is strict and replays fail, bound any client timestamp to a short window around now, and require the fields that payload type needs. Then store the accepted payload and republish the reference lists. A fixed-size feed payload holding a bounded number of newest-first references, oldest dropped, keeps reads cheap and predictable. Use a mutable batch, and plan for the node being down as an operational state rather than an outage: posts stop appearing, reading keeps working.

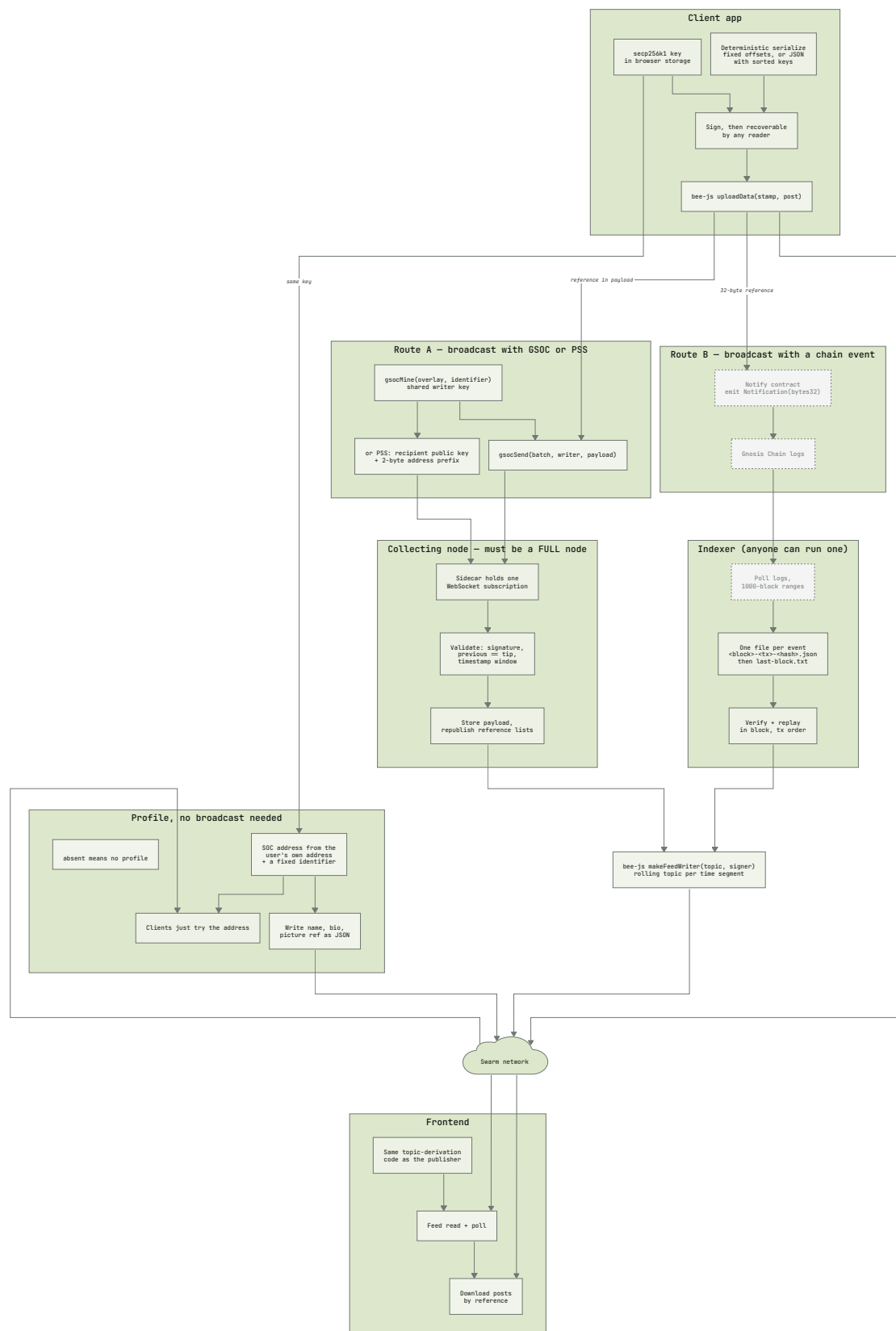
If you broadcast with a chain event. Deploy a contract whose body is `emit Notification(data)` with **Foundry (external)** on **Gnosis Chain (external)**, and call it with the reference the upload returned. Poll logs in ranges of at most a thousand blocks. Per batch, in this order: collect the events, write one file per event named `<block>-<txIndex>-<hash>.json` and skip any that already exists, write `last-block.txt`, then apply to in-memory state. That ordering is what makes a crash survivable, because on restart you resume from the larger of `last-block.txt` and the highest block number among the event filenames, and nothing is polled or applied twice. Take timestamps from the block rather than the post body, process same-block events in transaction-index order, and skip anything that fails validation instead of aborting the replay. Publish the indexer as something somebody else can run: same code, same inputs, same output, or the verification argument is decoration.

The read path. Publish rebuilt state with a feed writer, one feed for the index and one per thread. Share the topic-derivation function between publisher and frontend as literally the same code rather than two implementations of one rule, since a mismatch has the frontend reading an address nobody writes. Frontends poll on an interval, and a few seconds is fine for a conversation. During development, the subsidized `bzz.limo` gateway accepts uploads with a null batch ID and `outbox.wtf`

will emit the chain event over REST, so a first user needs neither BZZ nor gas to try the thing. Both are conveniences for getting started rather than something to design around.

TOOLS & REFERENCES

- bee-js library — <https://bee-js.ethswarm.org>
- Bee documentation — <https://docs.ethswarm.org>
- Multi-author blog: feeds referencing feeds, with runnable code — <https://docs.ethswarm.org/docs/develop/multi-author-blog>
- GSOC — <https://docs.ethswarm.org/docs/develop/tools-and-features/gsoc/>
- PSS — <https://docs.ethswarm.org/docs/develop/tools-and-features/pss/>
- Feeds — <https://docs.ethswarm.org/docs/develop/tools-and-features/feeds/>
- Postage batches, top-up and dilution — <https://docs.ethswarm.org/docs/develop/tools-and-features/buy-a-stamp-batch>
- bchan protocol: a GSOC message board specified byte by byte — <https://github.com/Cafe137/bchan-public>
- sounduncloud: the on-chain route, with the fullest written spec of it — <https://github.com/Cafe137/sounduncloud>
- **External dependencies (not Swarm primitives):** Gnosis Chain · Solidity · Foundry · viem or ethers



Component assembly. A client signs a post with a key held in browser storage and uploads it with bee-js, then takes one of the two broadcast routes: gsocSend to a collecting full node that validates the tip and republishes reference feeds, or a one-event contract call on Gnosis whose log any indexer replays into the same feeds. Profiles are written to a Single Owner Chunk address computed from the author's address, so clients read them without a lookup. External components are drawn dashed to mark them non-Swarm.

Reference implementations

Four prototypes, both broadcast routes, no production deployment. This space is not claimed: the projects below are worth reading for how they solved things and are not an implementation anybody has to work around.

bchan is an anonymous message board that broadcasts with GSOC and publishes with feeds, with no chain anywhere. Its protocol document is the most useful artifact in this space: exact byte layouts for both message types, the signed digest for each, the validation rules including the tip check, the feed topic scheme, and the reasoning behind it. It states its own limitation plainly, that the collecting node “is effectively the server”, while noting that the node operates entirely inside Swarm with no conventional backend beside it.

sounduncloud is a track-sharing hub for a web3 Strudel fork, where each track is Strudel code on Swarm that opens in the fork from a link. It broadcasts with an on-chain event, and its specification is where the deterministic JSON signing convention, the rolling-feed handover and the crash-recovery ordering in this document come from.

Manifesto Roulette is a proposed Devcon installation: self-contained HTML documents on Swarm with a permissionless comment layer beneath them, plus a physical control that triggers a real postage-batch top-up, so a visitor extends how long the collection and the responses around it stay funded. It is the clearest demonstration of community-funded persistence, because somebody can put their hand on it.

An earlier, private version of bchan broadcast with an on-chain event instead; the public protocol document supersedes it. Between them the two routes have now been built by the same author, which is part of why the choice reads as a genuine fork rather than a preference.