tinySSB Proof-of-Concept 4: Log Burning – Implementing Open-Ended Sessions with a Constant Number of Bounded Feeds

2022-04-18 christian.tschudin@unibas.ch

Abstract: In this PoC we show how subfeeds and feed continuations (see PoC-03) enable two peers to evolve trustworthy session state with an small number of feeds of limited length, although the data exchange is going on forever. Using a single append-only log clearly would not be able to do so as the log would become arbitrarily long. We introduce “feed hopping” and demonstrate this technique for sharing a symbol table in read-only mode. In the future, we plan to use the symbol table for compressing long feed IDs to small integer values.

The problem statement and use case of this Proof-o-Concept can be found in section 3.

1) Open Ended Session == “Sliding Window of Bounded Feeds”

Given two peers that mutually trust the other’s identity feed, we let each peer create a subfeed that serves to synchronize “session state”. These subfeeds will be mutually subscribed to and can also be trusted because a “subfeed creation event” represent a certificate about its trustworthyness.

In a second step, we assume that a subfeed will be terminated from time to time and at such times a continuation feed, called a segment, is declared and created. Each added session segment can also be trusted as there is a certificate chain from the root to the last segment in the chain. This configuration is shown in the following figure:

```
identity feed A (root)
  \sub
   `-> session feed A1 ..> A2 --> A3 --> A4
                       cont   cont   cont
                            <---sliding---->
                                window
```

Any peer which has not been in touch with A for some time can ask for all updates that occured during the separation and then catch up, including the signed “feed hopping” (feed continuation) events. This will let subscribers node perform a trusted replication of the full subchain. We will now focus on a scenario where a session is to be maintained between two specific peers A and B.

If node B can convey to node A that some old session feed segments are not needed anymore, node A can trim the chain of session feeds and only keep a “sliding window” of segments, also shown in the figure. Node B still can trust the content in the remaining logs because node B tracked the sliding window from the beginning and it even repeatedly allowed node A to prune old stuff.

However, if node B looses its knowledge about the first segment of the sliding window, or if A deletes (burns) old logs too quickly, there is no way to recompute the trust relationship starting from node A’s root feed because the “certificate chain” from the root to the last segment has a gap, by design. The width of the sliding window depends on the maximum round-trip-time between A and B as well as the maximum length of the feed segments. A bounded RTT will lead to a bounded sliding window.

1.1) Protocol for a Bidirectional Session Service over Append-only Logs

The figure below shows the two parties, A and B, which initially communicate with each other via two subfeeds A1 and B1. The subfeeds A1 and B1 form a bidirectional session between the two nodes, similar to TCP’s reliable, ordered delivery service.

```
         --.----------.                                                        -->t
Feed A1: ..|CONTDas A2|                  remove A1
         --'----------'                  ^
                 (1).------------.--     |                             -.------.--
Feed A2:            |CONTDfrom A1|..     |                            ..|ACK B1|..
                    `------------'--     |                        ^     -'------'--
  nodeA                |(2)              |                        |       |(6)
  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
  nodeB                |                 |(3)                     |(5)    |
                       v          --.------.--   ----------.      |       v
Feed B1:                          ..|ACK A2|..  |CONTDas B2|      |       remove B1
                                  --`------'--   ----------'      |
                                                       (4).------------.--
Feed B2:                                                  |CONTDfrom B1|..
                                                          `------------'--
```

At some point in time (1), node A decides to end the feed A1 and to continue communication via feed A2. The end message `continued-as A2` is put at the end of feed A1 while feed A2 is started with a `continued-from A1` message. When B receives the last A1 message, it signals to the network that it wants to subscribe to the new feed A2, and eventually, at time (2), gets the first A2 message. When all content of A1 has been processed, B puts an `acknowledge A2` message in its feed B1. On its reception, A knows that the switch to A2 is complete and that feed A1 can be removed ("log burning"), both from node A and from the network. Note that several "feed hopping events" can happen before `acknowledge A2` is received i.e., node A can decide to terminate its feed segents at its own pace regardless from how long it takes for B's acknowlegement to reach A.

The same sequence of events unfolds when B decides at time (4) to replace its feed B1 by a continuation feed B2. When A receives the `continued-as B2` message, it subscribes to B2. When A is done with processing the content of B1, it will acknowledge termination of B1 at time (6) and the network as well as node B can now remove feed B1 from their stores.

Such continuation events can be triggered by both sides concurrently and repeatedly: The reliable delivery and causality relationship between ending and continuing feeds makes sure that the signal messages are always received in the right order. Logically we have two "virtual content feeds", each being mapped to a sequence of real feed segments with the help of the three messages `continued-as`, `continued-from` and finally `ack`, for resource management reasons. Other resource management protocols are necessary, for example in a broadcast situation.

The nodes A and B must be able to keep track of at least four feeds: one current local feed plus a potential continuation segment, as well as the peer's current and one potential continuation segment. Besides these four+ feeds that represent one session, the nodes A and B have fixed identities which stay constant over time and serve for doing handshakes among themselves and with others. These constant IDs are used at peering time while the cycling feeds A1, A2, . . . (same for B1. . .) are the storage place for the messages that are part of the session between A and B. The overall feed dependency for a session is shown in the following figure:

```
identity feed A
  \
   `-> session feed A1 --> A2 --> A3 -->
                               ^
                               \mutual tracking of continuation events
identity feed B                 \and log burning after acknowledgements
  \                              v
   `-> session feed B1 --> B2 --> B3 -->
```

2) Feed Patterns for Resynching After Long Separation

The session implementation shown in the previous section is not useful in case two peers have been out of touch for long times. In this case, it is probably better to either create a new session from scratch, or to skip the many session messages that accumulated during the separation and potentially are not relevant anymore or not available anymore because the sending node had limited storage for its logs. How can one know the latest “session segment log” used by a peer, without having to, or being able to run through all feed continuation events?

```
identity feed A
  \sub \sub
   \    `-> session feed A1 --> A2 --> ... --> A2527
    \                      cont    cont   cont  ^
     \                                          |
      \                      .------------------'
       `-> head feed H1 --> H2
                       cont
```

One technique is to keep a “HEAD” subfeed which moves at slow speed compared to the session feed, like GitHub maintains a reference what currently is the head (of the master feed). This is shown in the figure above. Events in the head feed can happen in the (assumed seldom) event of peering. If peering events are happening often, then a two- or three-level hierarchy with feed continuation must be considered for having a slow-moving head-of-head-of-head feed.

Another technique applies when, after some years, even a multi-level head feed construct may become too long. In this case, the last ressort is to trust information conveyed at peering time (after a Diffie-Hellmann key exchange), outside of any append-only log. What we loose in this case is a track record of a peer’s advances in its head state and the assurance that a node had some specific past. In fact, the peer could have been reset to its initial state and any past interactions with it must be assumed to be lost. Note that the Internet’s web servers operate in such a “history-less” mode, so for many purposes this mode may be good enough.

Neither of the head feed techniques has been implemented for this Proof of Concept 4.

3) Synchronizing a Symbol Table for Arbitrarily Long Times with Four Bounded Logs

Our goal is to let two peers agree on a shared dictionary, also called a symbol table: if such a shared table exists, any reference to a symbol can be replaced by a small integer value that represents to index into that table. One use case is to communicate feed-specific information, for example the highest available sequence number, which means that in prinicple 32 Bytes are needed to reference the feed and then some more bytes for an integer value. Our goal is to bring the 34 to 36 uncompressed bytes down to three or five bytes, dependening on the number of symbols in the table and the length of the feeds.

In the Internet, a large family of “RObust Header Compression (ROHC)” techniques (see e.g. RFC5225) exist that can achieve impressive gains (1 header byte instead of 40 Bytes with plain IP). In our case of `tinySSB`, most headers have already been eliminated and been replaced by a 8 Bytes DMX field. Our interest is more related to compressing the 48 payload bytes, for example used by the replication protocol.

We will replicate a symbol table by streaming the potentially infinite sequence of table changes. This stream matches well the session service introduced in Sections 1 and 2.

3.1) Modelling a Symbol Table with Reference Counters

A symbol table enables to import long symbols (with several dozens of bytes) and to refer to them by small shortcuts (a few bytes for the offset into the symbol table). Such a table can also be used to find out whether a symbol was already encountered and that it therefore can be immediately replaced by its shortcut. Multiple applications can beneficially make us of one table in the case they refer to a sufficiently overlapping set of long symbols. After use, we will require applications to release a shortcut: using a reference counter, we can then internally decide whether the corresponding table entry can be removed for good or is still used by other apps. The following API is made available to the apps:

- `alloc(s)`: given an external symbol s, allocate a new shortcut or return the already allocated shortcut i and increase its reference counter
- `free(i)`: decrease the reference counter for the given shortcut i and release the binding if the reference counter drops to 0
- `find(s)`: given a symbol s, see if it has a shortcut - similar to alloc()
- `resolve(i)`: given a shortcut i, return the corresponding symbol

`alloc()` and `free()` are modifying the state of the symbol table while `find()` and `resolve()` are only querying it.

In our design, only one peer can modify the table while other peers keep a read-only replica, thus are limited to the last two methods. This read-only replication is captured in the following figure:

```
.----.    .----.
|app1| .. |appN|   <- refer to symbols via shortcuts ->  corresp. apps
`----'    `----'

alloc |  ^ find                                             ^ find
free  v  | resolve                                          | resolve
    .------.                                              .-------.
    |symtab|            --  updates -->                   |replica|
    `------'                                              `-------'
                                                          read-only
```

In a peering situation, each peer will have its own symbol table and the same symbol will have a different per-peer shortcut. Applications must therefore carefully distinguish local shortcuts from remote shortcuts received from peers.

The design of a jointly writable symbol table requires involved consensus mechanisms which we did not consider for this proof-of-concept. Instead, the "one-writer-multiple-readers" approach leads to a lean "update protocol" where the writer peer only needs three message types:

- `add X at Y`: define shortcut *Y* to be a placeholder for the long symbl *X*
- `remove Y`: undefine shortcut *Y*
- `resize SZ`: set the table size to the given number of entries

The PoC-04 demo can be started as follows:

```
% ./log_burning_demo.py init    # creates initial state for Alice and Bob
data
data/Alice
data/Alice/_blob
data/Alice/_logs
data/Alice/_logs/a6329aa106d64d33f861da0e5415688439dacd5f79ac7fb9f8803df0a1ca6cac.log
data/Alice/_logs/be45551db15d73aa4d1015bd18923e9e8b2dfaaa47d3d6c4847e3a5a2416b060.log
data/Alice/config.json
data/Alice/keystore.json
```

```
...
```

In separate terminal windows, start the symbol table-sharing apps for Alice and for Biob in any order (see below).

Alice will fill the symbol table with an initial set of ten entries and then periodically add a new random symbol and pick a random symbol to be removed. The table size thus remains constant over time but the stream of updates is infinite. Both peers limit their session feed segments to a maximum of nine entries. In the example below we see two “feed hopping events” from Alice. When running the peers for longer time one sees that Bob’s session feed (wherein he sends back acknowlegement messages) also is trimmed after 9 entries.

```
% ./log_burning_demo.py Alice
  creating face for UDP multicast group
  ...
Alice is be45551db15d73aa4d1015bd18923e9e8b2dfaaa47d3d6c4847e3a5a2416b060

Waiting for Bob to announce his subfeed ....
10:52:26.780    told to create subfeed for a6329aa106d64d33f861../2
10:52:26.782      new child is 46b868d31cacb37f78c9..
handshake received: remote sess feed is 46b868d31cacb37f78c9..

10:52:27.073  sess has started (catchup done, switching to live processing)
10:52:27.107  created shortcut 0 for symbol 0x4dcdfea52162c7e4ce88
10:52:27.131  created shortcut 1 for symbol 0x151d0247645ef491439a
10:52:27.149  created shortcut 2 for symbol 0x3c5c274f684847d87449
10:52:27.164  created shortcut 3 for symbol 0x07f1981cb38cd6565501
10:52:27.177  created shortcut 4 for symbol 0xc9183f61f86ade031096
10:52:27.190  created shortcut 5 for symbol 0x96ff480636bba7500a65
10:52:27.203  created shortcut 6 for symbol 0xab48df201405a3f304f7
10:52:27.203  SESS: ending feed c64e10f710ae438ee49a..
10:52:27.233    ... continued as feed 44d7932faf1a7e5e3c1d..
10:52:27.246  created shortcut 7 for symbol 0x05052d2f86b0690f5838
10:52:27.259  created shortcut 8 for symbol 0x46c6d001a4df0c9e75ba
10:52:27.271  created shortcut 9 for symbol 0xa5da549ba44e15b76fc1
10:52:27.284  created shortcut 10 for symbol 0x297677fddbc521f6efaa
10:52:27.296  removed shortcut 0
10:52:29.002  SESS: processing ack
10:52:29.002  SESS: removing feed c64e10f710ae438ee49a..
10:52:29.301  * Alice len(keystore)=2, len(dmxt)=4, len(openlogs)=4
10:52:29.336  created shortcut 0 for symbol 0x53d2501445d7a2fec9fb
10:52:29.359  removed shortcut 6
10:52:31.364  * Alice len(keystore)=2, len(dmxt)=4, len(openlogs)=4
10:52:31.365  SESS: ending feed 44d7932faf1a7e5e3c1d..
10:52:31.422    ... continued as feed 26c55418065a8cfa66c0..
10:52:31.441  created shortcut 6 for symbol 0x2ca7a64044ecd0d993d9
10:52:31.457  removed shortcut 10
10:52:32.096  SESS: processing ack
10:52:32.096  SESS: removing feed 44d7932faf1a7e5e3c1d..
...
```

```
% ./log_burning_demo.py Bob
  creating face for UDP multicast group
  ...
```

```
Bob is a6329aa106d64d33f861da0e5415688439dacd5f79ac7fb9f8803df0a1ca6cac

Waiting for Alice to announce her subfeed .
10:52:27.818    told to create subfeed for be45551db15d73aa4d10../2
10:52:27.821      new child is c64e10f710ae438ee49a..
10:52:27.923    told to stop old feed c64e10f710ae438ee49a../9
10:52:27.927    ... and to switch to new feed 44d7932faf1a7e5e3c1d..
handshake received: remote sess feed is c64e10f710ae438ee49a..

10:52:28.700  symtab notification: ['symbol added', '4dcdfea52162c7e4ce88', 0]
10:52:28.700  symtab notification: ['symbol added', '151d0247645ef491439a', 1]
10:52:28.700  symtab notification: ['symbol added', '3c5c274f684847d87449', 2]
10:52:28.700  symtab notification: ['symbol added', '07f1981cb38cd6565501', 3]
10:52:28.701  symtab notification: ['symbol added', 'c9183f61f86ade031096', 4]
10:52:28.701  symtab notification: ['symbol added', '96ff480636bba7500a65', 5]
10:52:28.701  symtab notification: ['symbol added', 'ab48df201405a3f304f7', 6]
10:52:28.735  symtab notification: ['symbol added', '05052d2f86b0690f5838', 7]
10:52:28.735  symtab notification: ['symbol added', '46c6d001a4df0c9e75ba', 8]
10:52:28.735  symtab notification: ['symbol added', 'a5da549ba44e15b76fc1', 9]
10:52:28.735  symtab notification: ['symbol added', 'cfae28074afb2a1f74a7', 10]
10:52:28.735  symtab notification: ['symbol remvd', '4dcdfea52162c7e4ce88', 0]
10:52:28.735  sess has started (catchup done, switching to live processing)
10:52:30.027  symtab notification: ['symbol added', '8aad05e092a0d5174703', 0]
10:52:30.046  symtab notification: ['symbol remvd', 'ab48df201405a3f304f7', 6]
10:52:32.030    told to stop old feed 44d7932faf1a7e5e3c1d../9
10:52:32.033    ... and to switch to new feed 26c55418065a8cfa66c0..
10:52:32.085  symtab notification: ['symbol added', '10e9f490bd7839ef0614', 6]
10:52:32.096  symtab notification: ['symbol remvd', 'cfae28074afb2a1f74a7', 10]
...
```

Note that Alice started to fill the symbol table immediately after the handshake with Bob was finished. Bob receives (at time `10:52:27.923`) a first feed continuation event from Alice that is processed by the session layer, but before the “symbol table app” had a chance to start. When finally the symbol table app is ready, it has to first digest the already delivered changes sent by Alice: after this catchup phase (= processing already received log entries), Bob switches to upcall-style processing where new updates are digested as they trickle in. The catchup phase is an example why it is necessary for Bob to signal to Alice (using an acknowledgement message) when an old feed can be removed. Another reasons is that the delivery of a session feed segment could have been delayed by the network without Alice knowing it, hence prematurely removing it could potentially lead to data and session synchronization loss.

4) Keeping State Across Application and Node Restarts

The above PoC app can be started once but fails if one stops and restarts either Alice's or Bob's side of the demo. The reason is that currently we do not persist all session state that the demo apps maintain while following the session's "sliding window". This relates to the following state:

- root ID
- initial session subfeed
- the chain of subfeed continuations

These items are stored twice on each node: once for the origin and once for the peer. Note that after the first session feed segment has been replaced and its replacement been confirmed, this information is dropped and only the active segments i.e., the sliding window, is kept in memory.

Additional state exists in form the secret keys associated with the session feed segments: a new keypair is created each time a subfeed is created (in our demo app this happens only once per node) and each time a continuation segment is created (this happens repeatedly). We have introduced a `keystore` module where new keypairs are stored and when a segment is removed, the associated keypair is dropped. To make sure that there is no resource leak we monitor from time to time the size of the `keystore` as well as the so called `DMX table` and the number of feeds in the local file repository. In the trace of the demo app shown above, two such information lines are visible for Alice, at time `10:52:29.301` and `10:52:31.364`.

4.1) Need for a Write-Ahead-Log (WAL)

In order to make our app restartable one would need to introduce a Write-Ahead-Log (WAL) where each node records the actions that it will make in the near future. From this, the app can reconstruct its internal state. As described above, this concerns the current sliding window of segments in the session module, the keystore module and the symbol table module.

We do not expand more on the requirement for this technique which is standard distributed systems engineering. In the reference Secure Scuttlebutt implementation (`sbot`), the WAL solution is simplified by having one append-only "journal" of all incoming log events in the order received, regardless of which feed they belong to. Any derived state can then reference the implied node-wide logical time (which is the length of the journal) and process any new events in the journal that arrived after the last recorded logical time (for some derived state).

In our PoC implementation we do not have that journal. Instead, each log is stored in a separate file which we want to have in order to delete individual session segments. The challenge thus is to create a node-wide logical clock that covers the many log files and their creation and deletion, as well as auxiliary information like secret keys generated on the fly. Creation, for example, of a continuation feed involves many steps like keypair generation, storage of the new crypto material on disk, allocation of a new feed file, signing and adding the `CONTDas` message to the old segment and the `CONTDfrom` message to the new segment. Because our demo app could be stopped at any moment in this sequence it must be able to find out where to pick up the work in order to complete the "create continuation feed transaction". For this PoC we have not done this and a stopped demo can only started from scratch i.e., with `./log_burning_demo.py init`.
