tinySSB Proof-of-Concept 3: Feed Trees and Log Dump Utility

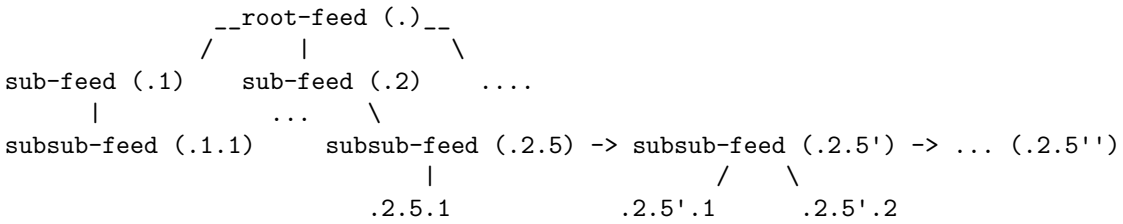
2022-04-09 christian.tschudin@unibas.ch

Abstract: We introduce four packet types that permit to expose feed relationships to the forwarding substrate. Two utilities have been written for feeds: one that permits to create a hierarchy of feeds based on a easy-to-write template (`ftree_make`), and another one to parse a persisted feed hierarchy and display its tree in graphical form (`ftree_load`). A third utility (`log_dump`) expands binary log files, including blob chains, and dumps their content in JSON format for easier debugging.

1) A tree hierarchy for dependent feeds

We introduce in tinySSB a way to express tree-shaped feed dependencies: a feed can have one or more children feeds and can end with an optional continuation feed. This is a special case of feed metadata where one feed exposes information about another feed. We aim at defining the minimal amount of meta data needed to form feed trees.

A parent-child relationship naturally forms a tree, extending it vertically. Horizontally, we introduce continuation feeds that extend a feed such that the tree shape is kept intact. The following figure shows these two extension possibilities (we attach classification-style names to feeds just for readability purposes):



We require that a child feed, as well as a continuation feed, specify their predecessor which is found either horizontally (if a continuation), or vertically (if a child node). The root feed has no predecessor, i.e. the respective predecessor feed ID is considered to be null.

As a result, it is always possible to find the root feed of the tree when starting from am arbitrary feed X like this:

```
find_root(x):
  while x.predecessor != null:
    x = x.predecessor
  return x
```

This is useful, for example, if trust claims are linked to the root feed and these trust claims should also apply to all dependent feeds.

The predecessor relationship is cryptographically protected such that no loops can form: Each dependent feed must provide a “birth-certificate” as a proof of its status, in its genesis block. This certificate consists of the feed ID of the parent, the sequence number where the child was declared, and the 12 last bytes of that log entry (which are thus taken from the signature field). Because it is possible that a parent feed declared the same child feed several times, either by accident or maliciously, this links a child’s first log entry to exactly one entry in the parent’s log.

Four packet types are introduced to encode feed relationships:

```
PKTTYPE_ischild   # a genesis block claiming that it has a parent feed
PKTTYPE_iscontn   # a genesis block claiming to be the continuation of a (now ended) feed
PKTTYPE_mkchild   # a declaration that a new child feed was created
PKTTYPE_contdas   # a declaration that this feed is continued as a new feed
```

Their use becomes visible in the steps (in pseudo code) needed to create a dependent feed:

```
mk_child_feed(parent):
  k = alloc_keypair()
  logEntry = parent.append(PKTTYPE_mkchild, k.pk, signed_with=parent.sk)
  f = create_feed(PKTTYPE_ischild, pred=(logEntry.fid/.seq),
                  proof=hash(logentry), signed_with=k.sk)
  return f

mk_continuation_feed(toBeContinued):
  k = alloc_keypair()
  finalLogEntry = toBeContinued.append(PKTTYPE_contdas, k.pk,
                                       signed_with=toBeContinued.sk)
  f = create_feed(PKTTYPE_iscontn, pred=(finalLogEntry.fid/.seq),
                  proof=hash(finalLogEntry), signed_with=k.sk)
  return f
```

To end a feed, a continuation entry is created for a dependent feed set to 0:

```
mk_end_of_feed(toBeEnded):
  finalLogEntry = toBeEnded.append(PKTTYPE_contdas, 0, signed_with=toBeEnded.sk)
```

2) Encoding

The presence of metafeed information is flagged by the packet's type. The metadata itself is stored in the packet's 48 bytes payload and varies among the four packet types:

```
PKTTYPE_ischild: (log entry must have sequence number 1)
  32B predecessor fid   # vertical (parent)
   4B predecessor seq
  12B hash(wirebits(fid[seq]))

PKTTYPE_iscontn: (log entry must have sequence number 1)
  32B predecessor fid   # horizontal
   4B prececessor seq
  12B hash(wirebits(fid[seq]))

PKTTYPE_mkchild: (at arbitray position in the parent log)
  32B child fid
  16B zeros (meaning: child is also a tinySSB feed)

PKTTYPE_contdas:  (log entry must be the last entry of the predecessor log)
  32B continuation fid
  16B zeros (meaning: child is also a tinySSB feed)
```

The zeroed field of the two last cases could be generalized to reference other-kinded feeds like classic SSB, SSB metafeeds, Bamboo etc., which permits to switch to a different log format. At this stage we

leave this further studies.

3) Discussion

In this section we quickly relate our tree-related primitives with those proposed in SSB's metafeed spec at →https://github.com/ssb-ngi-pointer/ssb-meta-feeds-spec.

The four packet types above form a minimal layer for expressing dependency in a way that enforces a tree shape. One can design a second layer of dependency, but without such guarantee, at application layer that uses ordinary plain48 or chain20 messages. In such 2nd layer dependency messages, things could be expressed like:

- mount (called `metafeed/add/existing` in the SSB metafeed spec)
- unmount (called `metafeed/tombstone` in the SSB metafeed spec)
- mounted (called `metafeed/announce` in the SSB metafeed spec)

The verb `mount` is borrowed from the UNIX vocabulary as it relates to extending tree-shaped file systems - and explains quite well the intent of the SSB metafeed design, we think. One could also have used the UNIX concept of symbolic links (`symlink(), unlink(), S_ISLINK()`) for this discussion. The point we want to stress is that these higher-level tree construction primitives can't prevent loop formation. The SSB metafeed spec does not discuss the problematic case where one metafeed would "add/existing" another metafeed, whether this desirable and how applications should deal with it.

SSB's `metafeed/add/derived` action is, tree-wise, already covered by our low-level `mkchild` action where the parent feed includes the child feed's ID. At the same time, the genesis block of the child feed links back to the respective `PKTTYPE_mkchild` log entry and proves the validity of this reference by including the hash of the mkchild log entry. This replaces both the two-fold signing (necessary for `add/existing` in SSB's metafeed spec) and the deterministic seed generation of a child's log, which in SSB's metafeed spec can be seen as a proof of belonging to some parent feed.

There remains the question whether the child's (or continuation's) keypair MUST be derived from information about the predecessor feed, instead of being created from a random seed. We chose to use a random seed in order to be able to drop a dependent feed's secret key for forward secrecy reasons. We think that this is an advantage over the SSB metafeed design. With the same intent we also prefer to *not* specify a special private message to oneself for storing the secret key of (any) dependent or top-level feed inside a feed (the SSB spec suggests a "metafeed/seed" log entry in the main feed), as this is an issue for which various techniques exist like key sharding that are unrelated to the specifics of feed trees.

4) A tool to create a feed hierarchy

The following program demonstrates the creation of a manually-defined feed tree, based on a template in form of a recursive dictionary data structure in Python. This template is currently hard-coded in the program and must be changed there. This could easily be replaced by reading it from a JSON file, in a next version of this proof-of-concept.

After writing the feeds to disk, the program outputs a visualization of the provided template (that can be later compared to a tree structure as extracted from feeds on disk, using the tool presented in the next section), as well as the JSON-formatted keystore where each feed's secret key is recorded.

Note that the keystore's sensitive content is only printed to stdout, thus putting the user in charge of grabbing the content and to store it in a safe place (which could be one of the feeds after encrypting them).

Templates are easy to adapt for specific needs. The template for the following example looks like this:

```
demo_feed_tree_template = {
    'name': 'root',
    'sub' : [
              {'name': 'apps',
               'sub' : [ {'name':'chat'}, {'name':'chess'} ]
              },
              {'name': 'main' }
            ]
}
```

As one can see, it is possible to annotate feeds with a name. This makes it easier to link a feed's public and secret keys with the feed's purpose (see the keystore output). When running the feed tree creation tool for above template, the following output is created:

```
% ./ftree_make.py data/tree
./ftree_make.py - 5 feeds created in 'data/tree/_logs'

--- visualization:

    / xx   'this is the root feed'
    + xx   'this is a sub feed'
    > xx   'this is a continuation feed'
    xx..   'feed can still be appended to'
    xx>>   'feed has a continuation'
    xx]]   'feed has ended'


  / root..
      + apps..
          + chat..
          + chess..
      + main..

--- keystore

{
  "d2698d9c8cb86036897c1cb61aed89bcea4d15092fae1905f7761aff07d16ce4": {
    "sk": "09add95f40623d0b30a3edfc46b8f1c34f199f70ec344d913d1d0857b832ec66",
```

```
    "name": "/root"
  },
  "62360fc983f70b228f1d2e24942ccf47724914b382d981d8e33c383158f401cb": {
    "sk": "ba0f64117cd13fd893d715adf2290ed97407d42d5b62bd81bcc5ac4af0dee898",
    "name": "/root/apps"
  },
  "f49930405e219b34516dcca55d9b4de7036846976e23b3859aa70c98f1c8e760": {
    "sk": "051c3a81b1cb6b78b57efd8acb72f0322578f3330ef372350f6a777f65506ec2",
    "name": "/root/apps/chat"
  },
  "4d42f97772521d225019695b03a61df4d67a50b524444e94a7ab8ec4178f89f6": {
    "sk": "1947db8199ed5f04bb14d4f52492de20c6a08c3178485b34027b3f74ebf6b683",
    "name": "/root/apps/chess"
  },
  "540c96911b70d1cfcff7b4a3ed9de92a9ffa28960fcad09b62dc48da752ad5f9": {
    "sk": "b819cf2ec483f0e97f110dbc89950ed798e2aefb7a61f4745f4b2993100c9926",
    "name": "/root/main"
  }
}
```

5) A tool to display a feed hierarchy

Once the feeds exist on disk, it is hard to understand their relationship as this involves extracting and interpreting binary packets from the log files. The following program does this work and starts from the given feed under the assumption that this is the root of a feed hierarchy.

```
% ./ftree_load.py data/tree/_logs/d2698d9c8cb86036897c1cb61aed89bcea4d15092fae1905f7761aff07d16ce4.lo
./ftree_load.py - display feed tree from repo at 'data/tree'

    / xx   'this is the root feed'
    + xx   'this is a sub feed'
    > xx   'this is a continuation feed'
    xx..   'feed can still be appended to'
    xx>>   'feed has a continuation'
    xx]]   'feed has ended'

  / d2698d9c8cb86036897c1cb61aed89bcea4d15092fae1905f7761aff07d16ce4..
      + 62360fc983f70b228f1d2e24942ccf47724914b382d981d8e33c383158f401cb..
          + f49930405e219b34516dcca55d9b4de7036846976e23b3859aa70c98f1c8e760..
          + 4d42f97772521d225019695b03a61df4d67a50b524444e94a7ab8ec4178f89f6..
      + 540c96911b70d1cfcff7b4a3ed9de92a9ffa28960fcad09b62dc48da752ad5f9..
```

Another tree with seven feeds, where some feeds have continuations and others are closed, would be displayed like this (one feed has been been continued 3 times and another one is declared ended):

```
  / a8657bd99ec2dee3a4e3a1545127afdbf2f948c255498e11c64ed4ae48044a44..
      + b1365c029ca6bb32877df1399fc227cc888727f6b0d1e5f02697e4c0921cf47b>>
          + dcf423d3ec8042d2c2ef579492d79ed8aa8f5652312d74e37c5edb2763a49103]]
          + 411b76266161363e17edf6bfd08b7bfaf5e7a54295ccbc82e08ea886509e41a6..
      > 683baf23a779c638e5be9bafeec97f7eca6a7046d9c87eecbffe085079de1e0e>>
      > 14b5f04d4877242c5f93327059bfbd23b2fde193f13c9aa5653f0acee4fd5dd3>>
      > 723687b2d447523cd2840f59f639cc875fb491187ac69c9f864711501d91b575..
```

When extracting feed relationship we have no name information at hand, unless one would introduce special log entries which “name” a feed’s purpose, which this program could then extract.

6) A tool to dump the content of log files and sidechain content

A special challenge is the interpretation of stored tinySSB packets because core attributes are not part of the wire bits. While the feedID is rather easy to extract (from the file name, or the log file’s management data), and the sequence number can be deduced from where in a file the packet was read from, the message ID `mid` of the previous packet (often called the “prev” field) is neither in the real packet nor in the log file: it can only be reconstructed by recomputing it, starting from the trust anchor information. Less complicated, but similarly tedious, is the collection of content bytes that are spread in a side chain.

Our dump tool reconstructs and displays these interrelated entities and displays them in JSON format. Currently, we try to be as complete as possible. For actual use, this proof-of-concept would need many configuration options for displaying single packets, their content in UTF-8, etc.

The following example is from a log that has two entries where the second entry has a sidechain with one blobs. Note that the *implicit* fields of a packet are marked with an asterisk. In case of the `dmx` value, both the implicitly computed value `dmx*` as well as the actual bytes from the packet `dmx` are shown in order to catch discrepancies.

```
% ./log_dump.py data/Alice/_logs/dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b8636c2a5e967d37d9e8ca6a3.log

{
  "dumptype": "tinySSB log file",
  "fid": "dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b8636c2a5e967d37d9e8ca6a3",
  "path": [
    "data/Alice/_logs/dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b8636c2a",
    "5e967d37d9e8ca6a3.log"
  ],
  "anchor.seq": 0,
  "anchor.msgID": "dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b",
  "front.seq": 2,
  "front.msgID": "629eb413215c7c47e77cdc9e241c3b171b804c3e",
  "store_length": 2
}

{
  "dumptype": "tinySSB log entry (packet)",
  "fid*": "dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b8636c2a5e967d37d9e8ca6a3",
  "seq*": 1,
  "prev*": "dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b",
  "name*": [
    "dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b8636c2a5e967d37d9e8ca6a3",
    "00000001dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b"
  ],
  "dmx*": "c5ee6d3d75157e",
  "mid*": "8c4b51422137f4c58c8607843123ae0301377375",
  "raw_len": 120,
  "raw": [
    "c5ee6d3d75157e006c6f6720656e747279203100000000000000000000000000",
```

```
    "00000000000000000000000000000000000000000000004bae274d057dc330",
    "4ac164f882c3ff33320c2b02eed3ccf314361269600435881f3adc4a4b9b6b18",
    "f24aca869c9493856d0b3636d58ad9575159a7a3af6c570b"
  ],
  "hptr": "b37c0e6b02e13113781925bd6d13565dbc9b7b22",
  "dmx": "c5ee6d3d75157e",
  "type": "0x00 = PKTTYPE_plain48",
  "payload": [
    "6c6f6720656e747279203100000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000"
  ],
  "signature": [
    "4bae274d057dc3304ac164f882c3ff33320c2b02eed3ccf31436126960043588",
    "1f3adc4a4b9b6b18f24aca869c9493856d0b3636d58ad9575159a7a3af6c570b"
  ]
}

{
  "dumptype": "tinySSB log entry (packet)",
  "fid*": "dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b8636c2a5e967d37d9e8ca6a3",
  "seq*": 2,
  "prev*": "8c4b51422137f4c58c8607843123ae0301377375",
  "name*": [
    "dd12d0c8b4aa5a7c3295d32fbf2fd9ea630df01b8636c2a5e967d37d9e8ca6a3",
    "000000028c4b51422137f4c58c8607843123ae0301377375"
  ],
  "dmx*": "b4e6d193a436a4",
  "mid*": "4f2c0baf0984bbf5caf620afe0a5e3e88af4f1b6",
  "raw_len": 120,
  "raw": [
    "b4e6d193a436a40151446174653a203136343934353333234380a546f3a202a0a",
    "5375626a9a8f7b38a0760ab8f25ee105a2580bc9c2a46a2dda1800edb0fb7848",
    "61a319d536eac9e16704a99193dcc35a60f6085fc735115cfb985f6e37233c5e",
    "a0839bc79966ce80e10d6efc47780542fcc568e3fa0f880e"
  ],
  "hptr": "a1c63ab633add35993ebe1d631a6faa1585185c4",
  "dmx": "b4e6d193a436a4",
  "type": "0x01 = PKTTYPE_chain20",
  "payload": [
    "51446174653a203136343934353333234380a546f3a202a0a5375626a9a8f7b38",
    "a0760ab8f25ee105a2580bc9c2a46a2d"
  ],
  "sidechain": {
    "content_declared_len": 81,
    "content_available_len": 81,
    "content": [
      "446174653a203136343934353333234380a546f3a202a0a5375626a6563743a20",
      "48692074686572650a0a7468697320697320d7920666972737420706f73742e",
      "204772656574696e67732c20416c696365"
    ],
    "first_blob": "9a8f7b38a0760ab8f25ee105a2580bc9c2a46a2d"
  },
```

```
  "signature": [
    "da1800edb0fb784861a319d536eac9e16704a99193dcc35a60f6085fc735115c",
    "fb985f6e37233c5ea0839bc79966ce80e10d6efc47780542fcc568e3fa0f880e"
  ]
}

{
  "dumptype": "tinySSB chained blob",
  "path": "data/Alice/_blob/9a/8f7b38a0760ab8f25ee105a2580bc9c2a46a2d",
  "hptr": "9a8f7b38a0760ab8f25ee105a2580bc9c2a46a2d",
  "content": [
    "6563743a204869207468657265650a0a74686973206973206d7920666972737420",
    "706f73742e204772656574696e67732c20416c696365000000000000000000000000",
    "0000000000000000000000000000000000000000000000000000000000000000",
    "00000000"
  ]
}
```
