Two Demos for the Python `tinyssb` library

2022-03-31 christian.tschudin@unibas.ch

Abstract: This README describes a test environment for tinySSB under UNIX, both for python3 and micropython (see also the note at the end).

Logical setup: Three nodes (nodeA, nodeB, nodeC) are simulated; the UNIX file system provides independent storage space; the nodes communicate via IP multicast. nodeA is the producer, the others are consumers. All nodes implement the "want" protocol (see below) where a single message type is used to request a specific log entry, which is sufficient to implement the reliable replication of append-only logs and synching incomplete log replicas.

The binary format of the log entries, which is the same as the wire format of the exchanged packets, is described in the packet spec *here* (FIXME).

1.a) Data is persisted in the `data` subdirectory

A directory called `data` is created where each node's logs, blobs and a config file are stored.

```
./data
    nodeA
       config.json
       _blob
          ...
       _logs
          ...
    nodeB
      ...
    nodeC
      ...
```

The Python libraries of `tinyssb` are at the same level as `data`, namely:

```
./tinyssb
    __init__.py
    dbg.py
    io.py
    node.py
    packet.py
    repository.py
    util.py

./pure25519    # an ed25519 implementation in pure Python
  ...
```

1.b) Initialization

Launch

```
% ./test_init.py
```

(or: `micropython ./test_init.py`) which creates above `data` directory structure.

2.a) Demo 1: Persisting a log, including side chain blobs

After initialization, launch

```
% ./test_repo.py   # run this several times
```

(or: `micropython ./test_repo.py`) which appends new log entries with each run and also dumps the log, either the 48 bytes of an ordinary entry, or the (arbitrarily long) content of a "sidechain entry":

```
...
[5] b'555555555555555555555555555555555555555555555555'
[6] b'666666666666666666666666666666666666666666666666'
[7] b'time.struct_time(tm_year=2022, tm_mon=3, tm_mday=31, tm_hour=18, tm_min=15, tm_sec=2, tm_wday=3
[8] b'888888888888888888888888888888888888888888888888'
...
```

See the source code in `repository.py` for the log format on disk.

2.b) Demo 2: One Producer and multiple Consumers

In one terminal window, initialize the `data` directory and start the producer:

```
% ./test_init.py
...
% ./test_prodcons.py
Producer/consumer demo for the tinySSB Python library
  creating face for UDP multicast group
    address is ('224.1.1.1', 5000)
Producer nodeA:
  starting thread with IO loop
Enter return to terminate...

18:18:07.426 install want callback @dmx=60ba77aacc0dc3
18:18:12.462 [2] enqueued for first time
...
```

The first time this producer is run, a genesis block is created for nodeA. In order to let the nodes nodeB and nodeC know about nodeA and trust nodeA's log, an empty replica log (for nodeA's log) is put into their `_log` directories, as a trust anchor. This empty log contains only information necessary to validate the first entry of nodeA's log, which nodeB and nodeC will have to request.

Now, in a second terminal window, you can start a consumer, for example nodeB:

```
% ./test_prodcons.py -cons nodeB
Producer/consumer demo for the tinySSB Python library
  creating face for UDP multicast group
    address is ('224.1.1.1', 5000)
Consumer nodeB:
  starting thread with IO loop
Enter return to terminate...

18:24:39.534 SND want request to dmx=60ba77aacc0dc3 for [1]
18:24:49.647 RCV new packet@dmx=8639555395a667, try to append it
18:24:49.670     log now at [1]
...
```

After having run the producer script once, you can start and stop the single producer, or one, or two consumer nodes, in arbitrary order, and in parallel or not. They will always synchronize, even transitively: nodeC can be offline while nodeA and nodeB have fully replicated content - then, once nodeC starts again and nodeA would in turn be offline, nodeC will receive all new content from nodeB.

This demo has only one producer and thus only one log is replicated.

2.c) Demo 3: Mutual Feed Replication

– not implemented yet.

This corresonds to a TCP-like setup (reliable bidirectional data stream) with the difference that it also works when there is only intermittend transitive connectivity.

3) Log Replication Protocol

The replication protocol is very simple: - a single request message is used, logically saying "I **want** log entry (feed=X,seq=Y)" - to which the corresponding entry `X[Y]` is returned, if a node has it.

This WANT message is regularly (10 sec) sent by a node wishing to replicate a log. It is formed by indicating the feedID and the sequence past the last one that was correctly received and stored. This enables to recover lost packets, thus implements ARQ.

The WANT messages with above numbering is also sent after each correctly received log entry, thus mechanically requesting subsequent messages. In this way, the same message type implements fast catchup (that could become necessary because either the producer or the receiver was offline for some time, it doesn't matter which one). Note that a replicating node can also send several WANT messages with consecutive numbers *in advance* in order to improve catchup time. This behavior is not implemented in the demo, though.

WANT messages have variable length (30-40B) while the log entries always are 120 bytes long.

The logging of reception events (showing also the number of received bytes) can be enabled in `node.py` by uncommenting the line that starts with `# dbg(GRE, "<< buf", len(buf), bin ...`

4) Blob Request Protocol

In the future there will be a second request message, for blobs: "I **need** blob 0xabcde...", to which a node that has a blob with a matching hash value will reply with that blob's plain 120 bytes. Here, an additional parameter "chained_credit" can be envisaged:

```
I need blob 0xabcde... and up to N followup blobs
```

which instructs the sender to extract from the end of the blob a hash value and also return that second blob etc, up to the requested limit.

Appendix

A) Note on `micropython` vs `python3`

If you initialize the data repository with running `test_init.py` using micropython, you must continue to use micropython (with `test_prodcons.py`), and vice versa for python3.

The micropython version changes the signature computation and verification: instead of the standard ed25519 signature, a keyed version of sha512 is used. That is, the binary logs in the log files are not compatible, which is why above note must be followed. See `mksignfct()` in `test_prodcons.py`

B) Communication means beyond Multicast

The default communication channel is via UDP multicast on 224.1.1.1/5000.

Three other channels exist:

- unicast UDP: send a UDP packet to a remote peer which takes note of the source address and from then on keeps a bidirectional channel
- serial/KISS: use a character device for sending and receiving tinySSB frames that are delimited with the KISS protocol / useful for interating with tinySSB's LoRa bridge

All interfaces can be added multiple times. If no interface is specified, multicast is added. For example

```
% ./test_prodcons -kiss /dev/tty.tinyssb-bridge -kiss /dev/tty.usbserial-0001
% ./test_prodcons -mc 224.1.1.1/5000 -udp 192.168.4.1/5001
% ./test_prodcons             # multicast to 224.1.1.1/5000 is added as a default
```
