(n+1)sec Test Report

(n+1)sec is a Free (libre), end-to-end encrypted, synchronous, multi-party messaging protocol and library, authored by eQualit.ie with support from the Open Technology Fund.

chats, (n+1)sec supports chats between groups of people. Like OTR, (n+1)sec is transport (n+1)sec is similar to Off The Record (OTR), but whereas OTR is limited to one-to-one agnostic, meaning it works on top of existing chat systems, such as XMPP or IRC. (n+1)sec can be implemented in any client and will work with all transports that meet two criteria: messages sent by clients are reflected back to them by the server; and all clients see all messages in the same order. The (n+1)sec protocol is described in this paper.

We produced two different sets of tests to help us develop (n+1)sec and to demonstrate its properties, giving them the collective name "EchoChamber." The first is a test harness and set of integration tests for the (n+1)sec library. These characterise the behaviour of the (n+1)sec library during normal use: so-called "black box" testing. Our black-box tests show how (n+1)sec behaves under various loads and networking conditions. We also produced a number of "white-box" tests. These investigate how (n+1)sec behaves when some users don't follow the protocol as expected. This might happen if someone were trying to deny service to honest users by using a modified version of the library to misbehave at the protocol level.

All of the tests generally focus on these criteria:

- How does (n+1)sec behave under a heavy load?
- How does it behave when certain events happen at specific or random times?
- What happens when some users don't respond as expected?

This document describes the various EchoChamber tests, reports the results of those tests and makes some conclusions about the performance of (n+1)sec.

Black-box testing

Our black-box test harness is written in Python and kept in a separate code repository from the (n+1)sec library. It is a testing platform for the (n+1)sec library that simulates network conditions and peer behavior to produce programmer-friendly benchmark data. The test harness spawns and controls multiple instances of our command-line (n+1)sec client Jabberite, passing arguments and reading output from it the same way a human user of that client would. This approach is easy to parallelise and simple to write.

The black-box tests are integration tests which demonstrates the (n+1)sec library working with other components to create a real-world result. Namely the connection of multiple chat clients to a chat server and the successful exchange of messages in an (n+1)sec encrypted channel.

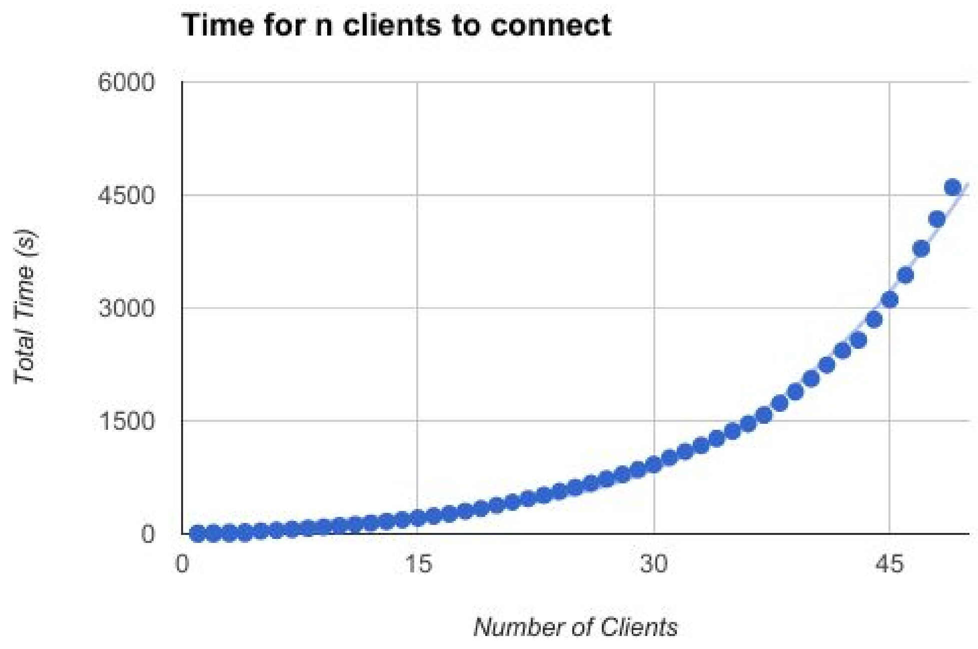
We used XMPP and the Prosody XMPP server as the chat message transport for these tests. However (n+1)sec is not dependant on any particular transport protocol. All tests were run on a virtual machine with a 6 core Intel Xeon L5639 (2.13 GHz).

To run the Python tests one needs to follow the instructions on the project's Github page [8].

Connection of clients

For this objective, we needed to demonstrate that the (n+1)sec protocol performed adequately when a large number of users are party to an encrypted conversation.

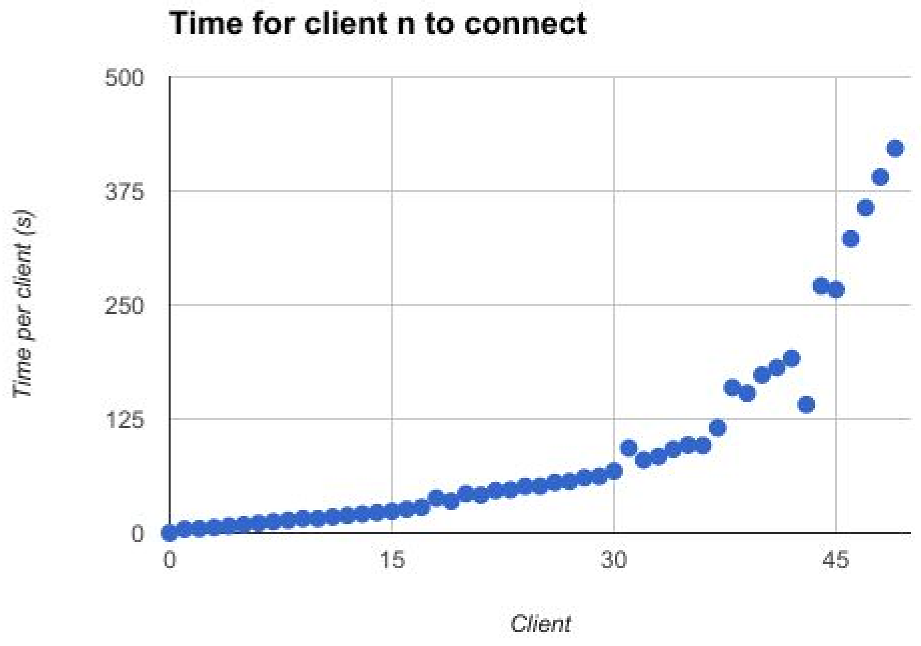
Python/test_client_connection: Here we instantialized N (= 2, 5, 50) jabberite clients and created a secure communication channel (conversation) between them.



In this test we measured the time taken for 50 clients to join a (n+1)sec conversation. The clients joined the conversation sequentially, one at a time. To join 50 clients sequentially to a conversation 50 key exchange rounds are necessary. These 50 rounds would require clients to perform a total of *1275* key exchanges (1st client performs 50 key exchanges, 2nd client performs 49 key exchanges etc.).

In the (n+1)sec protocol it is necessary for each participating client to take part in each key exchange to add a new client to the conversation. These leads to a polynomial time complexity which enforces a limit to the number of clients who can reasonably participate in one conversation.

This test shows that (n+1)sec library successfully supports 50 clients participating in a room. In this test all 50 Jabberite clients were running on a single 6 core virtual machine. The expensive step in the joining process is performing the key exchange computation which is CPU bound on our test machine. We believe that this key exchange would be significantly faster in a real-world environment where each user is only needs to perform one key exchange at a time.



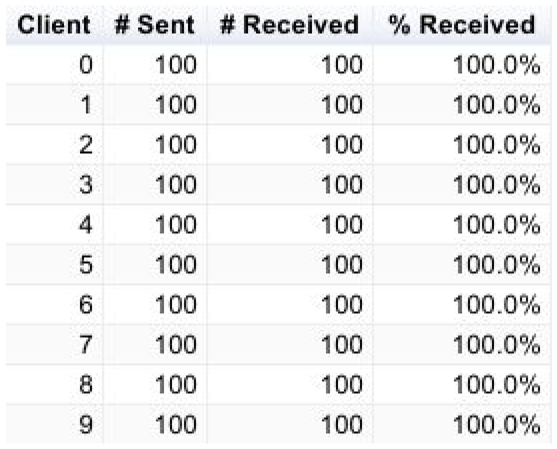
The above graph shows the time taken for the nth client to join an existing (n+1)sec conversation. As can be seen the change in time per client is non-uniform, particular for later clients. We believe this non-uniformity is an artifact of how our pexpect-based test harness interface with the Jabberite test client when running higher levels of CPU load.

Messaging

Python/test_messaging: As above, but also keep exchanging messages between the clients for for a defined period of time. In this test we used 10 and 25 clients.

10 Client Messaging Test

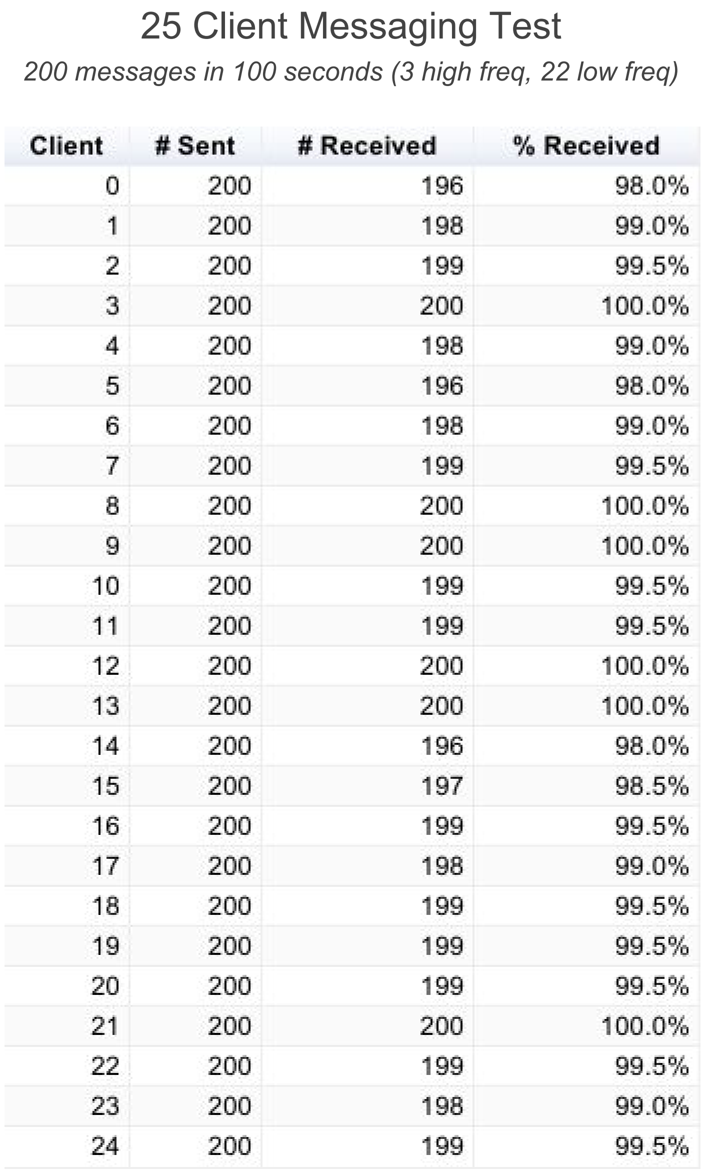
75 messages in 100 seconds (1 high freq, 9 low freq)



Client	# Sent	# Received	% Received
0	100	100	100.0%
1	100	100	100.0%
2	100	100	100.0%
3	100	100	100.0%
4	100	100	100.0%
5	100	100	100.0%
6	100	100	100.0%
7	100	100	100.0%
8	100	100	100.0%
9	100	100	100.0%

In this test 75 messages were sent between 10 clients with a random uniform distribution over the full time span of 100 seconds. One client was selected to represent a frequent message sending user and the remaining nine were low-frequency users. We choose this distributions to reflect the typical split between active and more passive participants in an online chat environment.

All clients received all of the messages which were sent during this test. This proves that (n+1)sec library can successfully performs its fundamental task. Multiple clients can join a conversation and exchange messages safely and securely.



25 Client Messaging Test

200 messages in 100 seconds (3 high freq, 22 low freq)

Client	# Sent	# Received	% Received
0	200	196	98.0%
1	200	198	99.0%
2	200	199	99.5%
3	200	200	100.0%
4	200	198	99.0%
5	200	196	98.0%
6	200	198	99.0%
7	200	199	99.5%
8	200	200	100.0%
9	200	200	100.0%
10	200	199	99.5%
11	200	199	99.5%
12	200	200	100.0%
13	200	200	100.0%
14	200	196	98.0%
15	200	197	98.5%
16	200	199	99.5%
17	200	198	99.0%
18	200	199	99.5%
19	200	199	99.5%
20	200	199	99.5%
21	200	200	100.0%
22	200	199	99.5%
23	200	198	99.0%
24	200	199	99.5%

Unfortunately, we have found bugs in a third-party library which we use in the Python implementation named `pexpect` [5]. This library serves as a communication link between the Python code and the Jabberite clients. We found that under heavy load this library dropped some of the events (mostly message receive events) and thus prevented the tests from finishing successfully. When we inspected the logs from each client, we found that all the events and messages were received correctly in jabberite, but got lost somewhere in pexpect. Later we found out that pexpect library has some reported and known issues similar (but not identical) to those we had been experiencing [6].

The test suite will mark some of the tests as failing due to the aforementioned bug in the pexpect library. To check whether jabberite clients are behaving as they should, one has to enable the debug mode using the --debug-mode flag and inspect the log of each client manually.

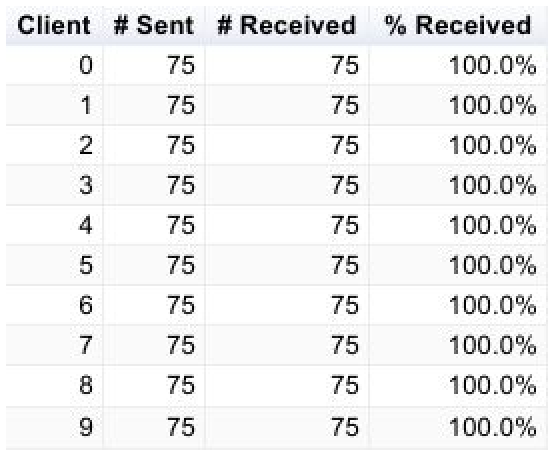
Adversarial and higher-latency network conditions

Python/test_messaging_high_latency: As above, but we added a transparent proxy server between clients and the XMPP server. The proxy server allows us to emulate the higher-latency network conditions associated with the use of censorship circumvention software and adversarial attempts to delay or drop messages.

10 Client Higher Latency Messaging Test

latency mean 0.2 s, variance 0.025 s

75 messages in 100 seconds (1 high freq, 9 low freq)



Client	# Sent	# Received	% Received
0	75	75	100.0%
1	75	75	100.0%
2	75	75	100.0%
3	75	75	100.0%
4	75	75	100.0%
5	75	75	100.0%
6	75	75	100.0%
7	75	75	100.0%
8	75	75	100.0%
9	75	75	100.0%

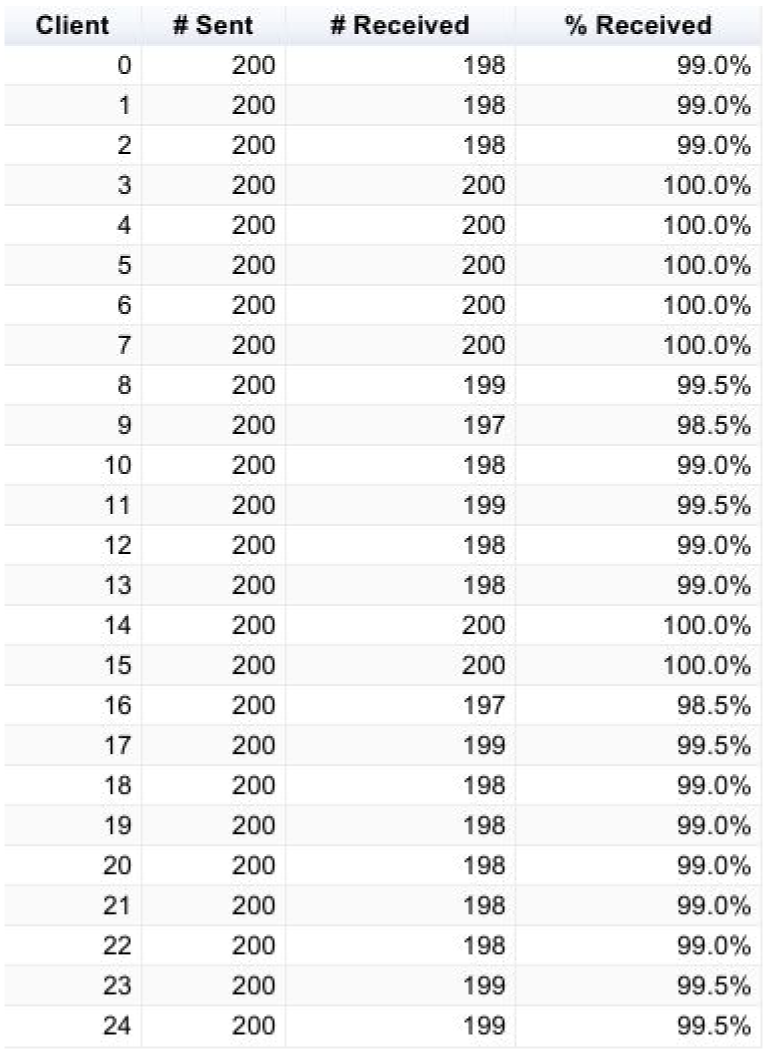
The higher latency messaging succeeds demonstrating 10 clients joining and exchanging 75 messages over the span of 100 seconds. We believe this test demonstrates that the (n+1)sec library is tolerant of network latency during the establishment and lifetime of a conversation.

As each client needs to exchange messages for each key exchange, the time taken to establish an (n+1)sec conversation will increase proportionally to the network latency of all participating clients. Once the conversation is established the network latency of any client should not significantly impact any of the other participants in the conversation.

25 Client Higher Latency Messaging Test

latency mean 0.2 s, variance 0.025 s

200 messages in 100 seconds (3 high freq, 22 low freq)



Client	# Sent	# Received	% Received
0	200	198	99.0%
1	200	198	99.0%
2	200	198	99.0%
3	200	200	100.0%
4	200	200	100.0%
5	200	200	100.0%
6	200	200	100.0%
7	200	200	100.0%
8	200	199	99.5%
9	200	197	98.5%
10	200	198	99.0%
11	200	199	99.5%
12	200	198	99.0%
13	200	198	99.0%
14	200	200	100.0%
15	200	200	100.0%
16	200	197	98.5%
17	200	199	99.5%
18	200	198	99.0%
19	200	198	99.0%
20	200	198	99.0%
21	200	198	99.0%
22	200	198	99.0%
23	200	199	99.5%
24	200	199	99.5%

The 25 client higher-latency messaging test encounters the same test harness issues as previously described for the messaging test. Under high-load conditions the test harness is dropping some messages resulting the < 100% message received rate.

We have confirmed that the issue lies in the Python interface to the Jabberite client and not in the underlying (n+1)sec library. Upon analysing the raw logs from the Jabberite clients we determined that each client had indeed received all of the sent messages.

White-box testing

While the black-box tests concentrate on how (n+1)sec behaves under heavy CPU and network loads, the C++ white-box tests make use of information from the server and / or internal state of the library to simulate how a malicious user would go about denying service to others.

The white-box tests use the (n+1)sec library directly and thus have a bigger repertoire of functions they can use. For communication they use the simplest possible TCP server running inside the same executable, which makes debugging easier and avoids bugs associated with inappropriate configuration of an XMPP server (as used with the Python implementation).

The white-box tests are part of the (n+1)sec project and thus are located in the same Github repository 9. To build them, one has to issue the command `make echo_chamber` explicitly as they are not built with the default `all` target. This is because the tests make use of additional Boost.Test and Boost.Asio libraries which we did not want to add as dependencies for the original library. To run all tests issue the command `./echo_chamber --log_level=test_suite`

Ratcheting

Under normal conditions, when a secure conversation is created, participants first exchange a secret which they then use for encrypting the messages that follow. About every three minutes this secret is recomputed for additional security. In the `test_ratcheting` test a malicious user constantly forces others to recompute the secret while they try to exchange messages at a high rate.

In this test each of three non malicious users attempts to send 100 messages and receive 100 messages from every other user. While doing so, a malicious user joins the channel and periodically requests a key exchange inside the conversation trying to de-sync the other three users.

Such message exchange would normally only happen if a member of the conversation has joined/left or after a predefined timeout (about 200 seconds).

Key exchange is one of the more complex tasks users need to do and is tricky to get it done correctly in the code. Despite that, the test has not exposed any bugs, thus giving us some confidence in this part of the code.

Message dropping

A malicious user could try to prevent a conversation from starting by refusing to respond to requests made by other clients according to the (n+1)sec protocol. The expected behavior in such a case is that the malicious user will be excluded from the conversation and non-malicious users can continue their discussion without them. This behavior is tested in five tests belonging to the `test_drop_message` family.

Here we first create a conversation with three non malicious and one malicious participant. Once the conversation is created a new client attempts to join while the malicious one keeps ignoring (fails to respond to) messages significant to the inclusion of the new client.

The test consists of five different sub-tests where in each the malicious user ignores a different message type.

Each of the sub-tests passes with success which means that the new client successfully joined the conversation while the malicious one was rejected.

Invitations

These tests vary the timings with which clients send messages to the server during different stages of the conversation lifecycle. The intent is to check whether the library is sensitive to timing race-conditions.

The scope of these nine tests is similar to the that of the test_client_connection black-box test, but instead of only appending each new user to an existing secure conversation one by one, we have a number of joining strategies depending on the number of clients, joining timeouts, and whether the join should happen in sequence or in parallel.

The invitations tests identified two protocol edge-cases that the library failed to handle correctly. Both bugs have been fixed.

Session join order

If two or more people join an encrypted conversation at the same time, the message exchanges required to bring each into the conversation occur interleaved with each other, leading to potential race conditions in the protocol.

Under normal circumstance a participant who just joined a session would be presented with a list of users already participating, and thereafter would receive user join events only on users not already present. While working on other EchoChamber tests, we exposed a bug where this was not the case, presumably due to the concurrent joining of users. Some of the clients received events that claimed users already present in the conversation joined again.

This test creates the above situation deliberately in order to examine more precisely how the protocol behaves. We prepare a conversation where users join concurrently and then test what happens when we add one more client.

The bug this test exposes has not yet been fixed and is documented here.

Message exchange

These tests add to the invitations tests by including the sending and receiving of user messages with various timings. Once again the intent is to check whether the library is sensitive to timing race-conditions.

Both tests create 10 clients which then exchange 30 messages. The tests differ in the way they send the messages. While in the simpler version each client sends a message only after it has received its previous one, in the more complex case all clients send messages after random duration has elapsed.

The scope of these two tests is similar that of the black-boxtest_messaging black-box test, and were written to investigate a bug exposed by that black-box test (with success, the fix can be found here).

Discussion

Writing these test gave us a solid understanding on where the (n+1)sec library stands in terms of stability and resilience against malicious actors. We have identified 6 bugs which would have been next to impossible to reproduce using only manual testing and even harder to debug. We managed to fix three ([1], [2], [3]) and left the rest as a future work by documenting them on the Github issues page [4].

The bugs we did find and managed to fix were all in the category of "missed corner cases". The ones we've not yet fixed as well seem to belong to that category but a closer look needs to be taken. On the other hand the library showed to be robust against malicious attack vectors.

As it goes with software development, new corner cases may show up (although with decreasing probability) and new attack vectors can be thought of, in which case new tests should be added.

While running the Python tests (which parallelize by default) we found that each jabberite instance was utilizing CPU cores to the maximum. For example, a test with four nodes on a four core CPU would utilize the whole CPU to 100%. When profiling, the program seems to have been mostly in the cryptographic parts of the code which are (of course) essential. However, we think that in practice CPU won't pose usability problems as it will be unusual for a user to run more than one (n+1)sec client and the rate of message exchange shall be considerably lower than those used in the tests.

A more serious bottleneck (also apparent from the tests where many clients were used) is that of the number of message exchanges during a session creation. That is, each time a new client starts joining a conversation it needs to exchange a constant number of messages with other participants, given that each message sent to the server needs to be echoed back to O(n) participants, and each of those participants needs to respond with a message which again gets echoed to O(n) participants, we have that one client joining requires O(n^2) messages sent by the server. Further, knowing that each of the nodes in a conversation of size n has to do this, we have that creating such conversation requires the server to send O(1^2 + 2^2 + ... + n^2) = O(n^3) messages.

To the best of our knowledge, this bottleneck is implied by the cryptographic features (n+1)sec provides (end to end encryption, forward secrecy and plausible deniability) as we are not aware of existing algorithms providing asymptotically better network performance. While it is certainly an aspect worth investigating in future work, It can also be argued that increasing the number of participants in a channel decreases its security simply because there are more people to potentially leak information within it.

References

[1] https://github.com/equalitie/np1sec/commit/976a93ac6255d0b8a88b9d7686dfb0c00f493445
[2] https://github.com/equalitie/np1sec/commit/c8e52fad3f18f5260f41ee14da92b000ce97c66d
[3] https://github.com/equalitie/np1sec/commit/ac2ab49a84c3971a1cb5a4f3823f0e0755026d3a
[4] https://github.com/equalitie/np1sec/issues
[5] https://pexpect.readthedocs.io
[6] https://pexpect.readthedocs.io/en/stable/commonissues.html
[7] https://github.com/equalitie/np1sec/issues/47
[8] https://github.com/equalitie/EchoChamber
[9] https://github.com/equalitie/np1sec