Testing

Re-listed Low Level Requirements

The list of requirements low level requirements, identical to that in the Analysis section.

RSA Implementation (A)

1. A reliable interface to a hashing module
2. RSA Key Handling:
    1. Be able to load RSA public/private keys in PEM format from files / STDIN
    2. Be able to validate the format of these keys
    3. Be able to parse these keys extracting all necessary numbers for RSA decryption
3. Signing/Verification
    1. Have a valid RSA encryption scheme (encryption with public key)
    2. Have a valid RSA decryption scheme (decryption with private key)
    3. Have a valid RSA signing (sig) scheme (signing with private key)
    4. Have a valid RSA signature verification (verif) scheme (verify with public key)
4. Object Signing
    1. Algorithm to convert an object to a hash in a reproducible way, minimising the chance of hash collisions
    2. Ability to sign a class of object with RSA sig scheme
    3. Ability to verify a signed object with RSA verif scheme, raising an error if signature is invalid

Debt Simplification (B)

1. A reliable digraph structure, with operations to
    1. Get the nodes in the graph
    2. Check if an edge exists between two nodes
    3. Nodes can be added
    4. Nodes can be removed
    5. Edges can be added
    6. Edges can be removed
    7. Neighbours of a node should be easily accessed (neighbours for the purposes of a breadth first search)
2. A reliable flow graph structure
    1. All the operations listed in B.1.1

    2. Adding an edge should have different functionality: edge should be able to be added with a capacity, and edges should have a notion of flow and unused capacity
    3. Be able to return neighbours of nodes in the residual graph (i.e. edges, including residual edges, that have unused capacity)
    4. A way to get the bottleneck value of a path, given a path of nodes

3. A reliable recursive BFS that works on flow graphs

4. Implementation of Edmonds-Karp

    1. Way to find the shortest augmenting path between two nodes
    2. Way to find bottleneck value of a path
    3. Finding max flow along a flow graph from source node to sink node

5. Simplifying an entire graph using Edmonds Karp, using the method laid out in Settling a graph using a Max Flow algorithm.

6. Be able to convert a list of valid transactions into a flow graph

7. Be able to convert a flow graph into a list of transactions, signed by the server

8. Be able to simplify a group of transactions, having each transaction individually verified before settling

Client / Server Structure (C)

1. The server should be accessible to the client via a REST API
2. The client should be relatively thin, only dealing with input from user and handling error 400 and 500 codes gracefully.
3. Client and Server should communicate over HTTP, using JSON as an information interchange format
4. The client should have a clear, easy to use command line interface

'Integrated' requirements for how the end system should behave (D)

1. Ensuring the validity of transactions

    1. If a transaction is tampered with in the database, it should be classed as unverified
    2. A user should not be able to sign an already signed transaction
    3. A user should not be able to sign a transaction where they are not one of the listed members
    4. A user should not be able to sign a transaction with a key that is not associated to their account
    5. A user should not be able to sign a transaction without entering their password correctly
    6. Every time a transaction is pulled from the database and sent to the user, it should be verified by the server using the RSA sig/verif scheme from section A

2. Ensuring that the debt simplification feature works
    1. All transactions in the group being settled should be verified upon being pulled from the database
    2. It should not be possible to simplify a group if there are unverified transactions in the group
    3. If the transaction structure of the group does not change, the user should be notified
    4. The simplification should accurately simplify a system of debts such that no one is owed / owes a different amount of money after simplification
    5. The simplifying process should result in unverified transactions being produced, able to be signed by the user
3. Ancillary features
    1. Users should be able to register for an account, providing name, email, password and a PEM formatted private key
    2. Users should be able to create transactions where they are the party owing money; these transactions should be created as unsigned
    3. Users should be able to create a group with a name and password
    4. Users should be able to join a group by group ID
    5. Users should be able to mark a transaction as settled; transactions should only be marked as settled when both parties involved mark the transaction as settled
    6. Users should be able to see which groups they are a member of
    7. Users should be able to see all of their open transactions
    8. Users should be able to see all the open transactions in a group (whether they are part of the group)
    9. Users should be able to see the public key information of any user on the system
    10. Users should be able to see individual transactions by passing in a transaction ID

Database Architecture (E)

1. User information
    1. User ID
    2. Contact info
    3. A hash of the user's password
    4. Associated Groups
    5. Public Key (provisions for one or more)
2. Transaction Information
    1. Transaction ID
    2. Payee
    3. Recipient
    4. Transaction reference
    5. Amount (£)
    6. Payee's signature
    7. Recipient's signature
    8. Whether transaction has been settled

3. Group information
    1. Group name
    2. Group password
    3. People in the group
    4. Transactions in the group

Unit Test Framework

To demonstrate the effectiveness and completeness of sections A & B, I will provide my unit testing framework. I approached implementation from a test-driven development perspective, and thus all of these tests were written before the code they run was implemented.

This has led to the creation of an extensive, robust framework of tests, which effectively shows the extent to which I have completed sections A and B of my project.

The test harness has ~80% coverage on the `transactions`. `simplify` and `crypto` modules, with the crypto module having >90% coverage. This makes it hard to argue that my solution has not been adequately tested.

Below is a report of my tests running generated by my IDE, as well as a table linking individual unit tests to requirements from sections A and B. An interactive version of the file is included in the project files.

The unit tests are provided after the Evaluation.

(Note - the slightly long time that these tests took to run can be attributed to the drawing of graphs. I used the `graphviz` library to dynamically generate pictures of graphs of the debt that my algorithm was simplifying. Some of these are included below.)

: 100 total, 100 passed

2.45 s

Collapse | Expand

- test_crypto — 600 ms
  - test_hashes — 11 ms
    - TestHash — 11 ms
      - test_hash (tests that hash object can validate hash looking numbers) — 9 ms
        - [valid] — passed
        - [too short] — passed
        - [too long] — passed
        - [wrong type] — passed
      - test_hasher_fails (Checks that hasher objects when not bytes are passed into it) — passed — 0 ms
      - test_init (Checks that _hasher is initialised correctly i_e_ no strange start values) — passed — 0 ms
      - test_update_digest (Ensures that a hash with a given value will digest the correct thing) — 2 ms
        - [before update] — passed
        - [after update] — passed
  - test_keys — 377 ms
    - TestRSAKeyLoading — 377 ms
      - test_file_loading (Tests that file is being loaded correctly assuming correct file) — passed — 99 ms

        /home/tcassar/projects/settle/src
        0
      - test_file_not_found — passed — 7 ms

        /home/tcassar/projects/settle/src
        0
      - test_parsing — 60 ms
        - [pub_exp] — passed
        - [priv_exp] — passed
        - [mod] — passed
      - test_public_key — 101 ms
        - [allowed access] — passed
        - [deny access] — passed
      - test_unloaded_key — passed — 7 ms

        /home/tcassar/projects/settle/src
        0
      - test_unparsed_key (Check accessing attributes) — passed — 58 ms

        /home/tcassar/projects/settle/src
        0
      - test_wrong_format (Checks that we can deal with files being the wrong format) — passed — 45 ms

        /home/tcassar/projects/settle/src
        0
        unable to load Private Key
        140267057518400:error:0909006C:PEM routines:get_name:no start line:../crypto/pem/pem_lib.c:745:Expecting: ANY PRIVATE KEY

Collapse | Expand

- test_sign_verify — 212 ms
  - TestRSA — 212 ms
    - test_RSA_sign (Checks for consistent creating / verifying of a 'signature') — passed — 108 ms
    - test_encryption (Checks to see if process is reversible, and encrypted is different to how it started) — 104 ms
      - [Catch Public Key] — passed
      - [encrypted] — passed
      - [Successful decryption] — passed

- test_settling — 2.92 s
  - test_Path — 9 ms
    - TestPath — 9 ms
      - test_build_bfs_struct — 3 ms
        - [with initial value] — passed
        - [with initial value] — passed
      - test_build_path (Given a prev_map, check we build the right path) — passed — 0 ms
      - test_find_target — passed — 1 ms
      - test_recursive_bfs (Check that BFS is finding paths along graph correctly) — passed — 1 ms
      - test_shortest_path (Checks that we can in fact find shortest path along) — 4 ms
        - [digraph] — passed
        - [weighted_graph] — passed
        - [flow] — passed
  - test_flow — 2.90 s
    - TestFlowEdge — 4 ms
      - test_adjust_edge — passed — 1 ms
      - test_push_flow (Checks that we update flow by required amount) — 2 ms
        - [R: 1 [0/0],] — passed
        - [1 [3/5],] — passed
        - [exceed capacity] — passed
      - test_unused_capacity (builds two edges, residual and non-residual) — 1 ms
        - [R: 1 [-3/0],] — passed
        - [1 [0/5],] — passed
    - TestFlowGraph — 522 ms
      - test_add_edge — 370 ms
        - [negative test] — passed
        - [added] — passed
        - [Net debt (adding)] — passed
        - [Net debt (removing)] — passed

test_bool passed 0 ms

{Vertex(ID=0, label='a'): [], Vertex(ID=1, label='b'): [], Vertex(ID=2, label='c'): [], Vertex(ID=3, label='d'): [], Vertex(ID=4, label='e'): []}

test_flow_neighbours (Checks we get edges that have unused capacity, including residual) passed 143 ms

DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph0'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph0']

test_get_edge passed 0 ms

test_is_edge 1 ms

- [no edge] passed
- [edge] passed

test_pop_edge 8 ms

- [negative] passed
- [removed edge] passed
- [removed residual] passed

TestMaxFlow 514 ms

test_augment_flow 172 ms

- [a -> b] passed
- [b -> c] passed
- [c -> d] passed
- [d -> c] passed
- [c -> b] passed
- [b -> a] passed

test_augmenting_path passed 0 ms

test_bottleneck passed 154 ms

DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph0'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph0']

test_edmonds_karp passed 181 ms

DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph0'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph0']

test_nodes_to_path passed 3 ms

test_old_edmonds passed 4 ms

TestSimplify 1.86 s

test_adjust_edges passed 588 ms

DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph0'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph0']
DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph1'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph1']

Collapse | Expand

DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph1']
DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph4'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph4']

test_mithun_simplify passed 391 ms

DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph0'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph0']
DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph1'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph1']

test_netflow (Checking people owed same before and after) passed 0 ms

test_simplest_graph (Shouldn't change) passed 496 ms

DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph0'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph0']
DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph1'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph1']

test_simplify_debt passed 389 ms

{Vertex(ID=0, label='d'): 15, Vertex(ID=1, label='m'): 0, Vertex(ID=2, label='t'): -15}
DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph0'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph0']
DEBUG:graphviz._tools:os.makedirs('./graph_renders')
DEBUG:graphviz.saving:write lines to './graph_renders/graph1'
DEBUG:graphviz.backend.execute:run [PosixPath('dot'), '-Kdot', '-Tsvg', '-O', 'graph1']

test_graph 6 ms

TestDigraph 5 ms

test_add_edge passed 1 ms

test_add_node passed 0 ms

test_init 1 ms

[init] passed

[helpers] passed

test_is_edge 2 ms

[is edge] passed

[isn't edge] passed

test_nodes passed 0 ms

test_pop_edge passed 0 ms

test_pop_node passed 1 ms

U -> VW
V -> W
W ->
V -> W
W ->

TestWeightedDigraph 1 ms

test_add_edge passed 0 ms

Collapse | Expand

- test_add_edge — passed — 0 ms
- test_add_existing_edge — passed — 0 ms
- test_flow_through — 1 ms
  - [u] — passed
  - [v] — passed
  - [w] — passed

test_transactions — 3.93 s

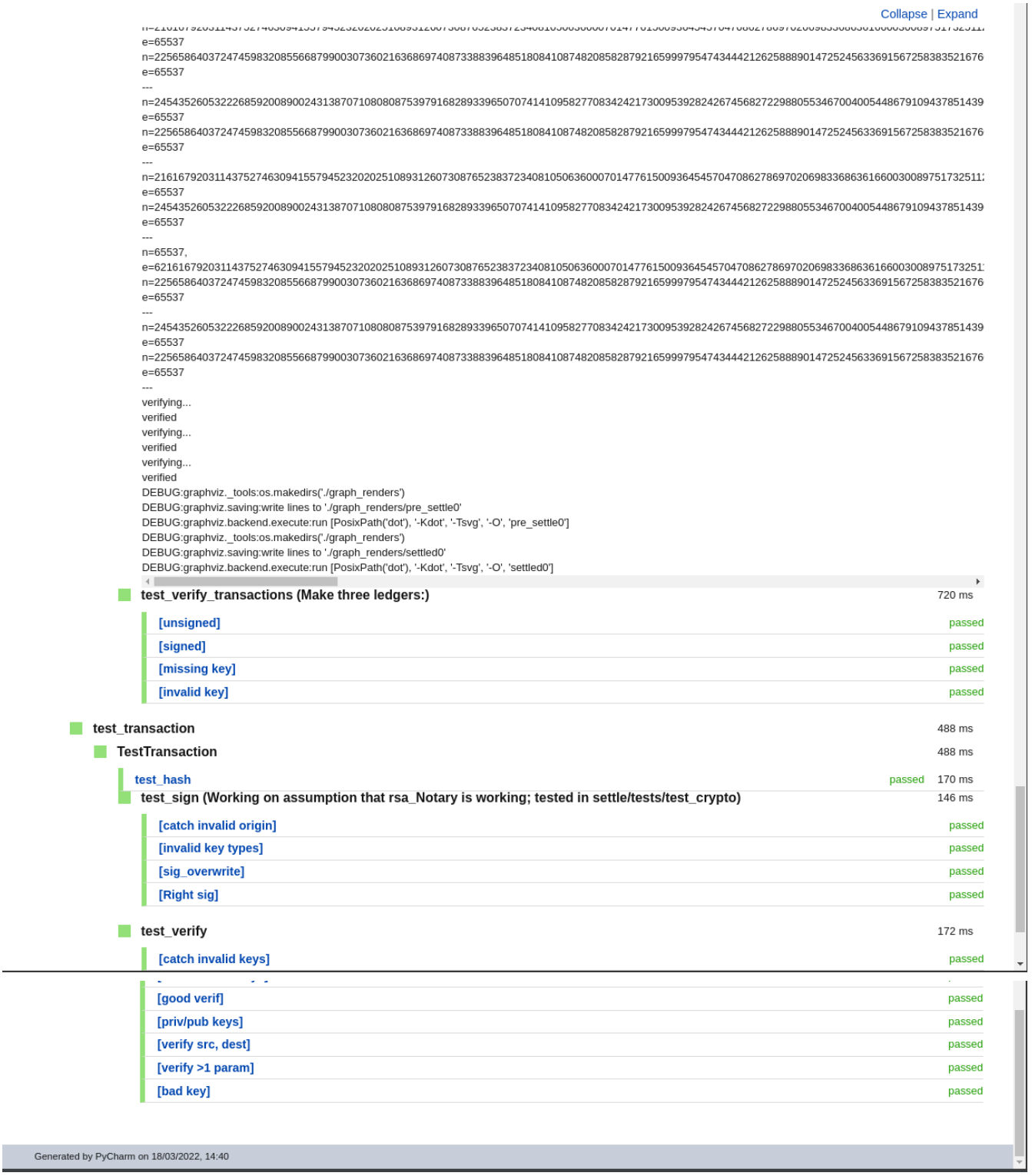
- test_ledger — 3.44 s
  - TestLedger — 3.44 s
    - test_add — 234 ms
      - [Add] — passed
      - [Catch non transaction] — passed
    - test_as_flow — 715 ms
      - [nodes] — passed
      - [to flow graph] — passed
    - test_flow_to_transactions — 551 ms
      - [to transactions] — passed
    - test_load_from_csv — passed — 285 ms

loading from csv
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514392
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216762
e=65537
---
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514392
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216762
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514392
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216762
e=65537
---
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514392
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216762

e=65537
---
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514390
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216760
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514390
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216760
e=65537
---
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514390
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216760
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514390
e=65537
---
n=65537,
e=6216167920311437527463094155794523202025108931260730876523837234081050636000701477615009364545704708627869702069833686361660030089751732511
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216760
e=65537
---
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514390
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216760
e=65537
---

test_simplify_ledger passed 937 ms

loading from csv
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514390
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216760
e=65537
---
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514390
e=65537
n=2256586403724745983208556687990030736021636869740873388396485180841087482085828792165999795474344421262588890147252456336915672583835216760
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112
e=65537
n=2454352605322268592008900243138707108080875397916828933965070741410958277083424217300953928242674568272298805534670040054486791094378514390
e=65537
---
n=2161679203114375274630941557945232020251089312607308765238372340810506360007014776150093645457047086278697020698336863616600300897517325112



Evidence of meeting Requirements - Section A & B

Crypto (A)		File	Test(s)
A1		test_crypto.test_hashes	*
A2		test_crypto.test_keys	
	A2.1		test_file_not_found test_file_loading
	A2.2		test_wrong_format
	A2.3		test_parsing test_unparsed_key test_unloaded_key test_public_key
A3			test_sign_verify
	A3.1		test_encryption
	A3.2		test_encryption
	A3.3		test_RSA_sign
	A3.4		test_RSA_sign
A4		test_transactions.test_transaction	
	A4.1		test_hash
	A4.2		test_sign
	A4.3		test_verify
Simplify (B)		test_settling	
B1		test_graph	
	B1.1		test_nodes
	B1.2		test_is_edge
	B1.3		test_add_node
	B1.4		test_pop_node
	B1.5		test_add_edge
	B1.6		test_pop_edge
	B1.7		
B2		test_flow.TestFlowGraph	
	B2.1		tests by the same name as above
	B2.2		test_add_edge
	B2.3		test_flow_neighbours
	B2.4		test_bottleneck
B3		test_Path	
	B3.1		All tests in file
	B3.2		All tests in file
B4		test_flow.TestMaxFlow	
	B4.1		test_augmenting_path
	B4.2		test_bottleneck
	B4.3		test_nodes_to_path test_augment_flow test_edmonds_karp
B5		test_flow.TestSimplify	
			All tests in TestCase
B6		test_transactions.test_ledger	test_as_flow
B7			test_flow_to_transactions
B8			test_verify_transactions test_simplify_ledger

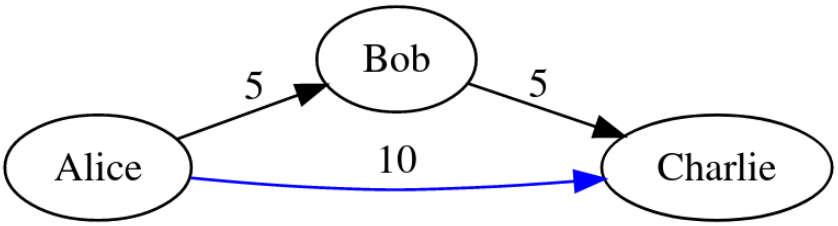
I appreciate this table may not show information about the individual tests. Every unit test I wrote has an informative docstring. Thus, refer to the code after the Evaluation for more clarification about the function of any unittest.

Proof of simplification algorithm functionality (B5)

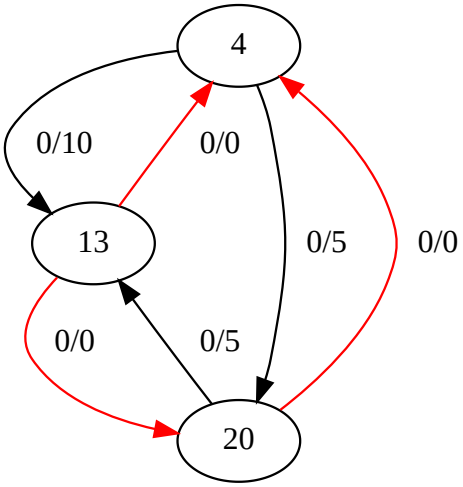
In the analysis section, I provided a hand-trace of the expected graph structures involved in settling a system of debts. Having done this hand tracing, I decided to use the same graph to test my algorithm.

Using the `graphviz` library I was able to generate before and after representations of the flow graphs used to simplify the system of debt it was fed. Here is my initial structure compared to the expected structure

Expected



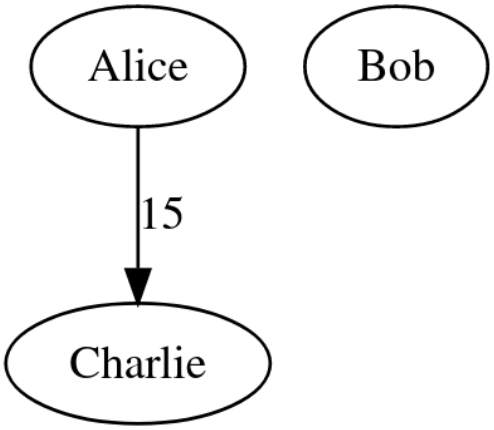
Generated



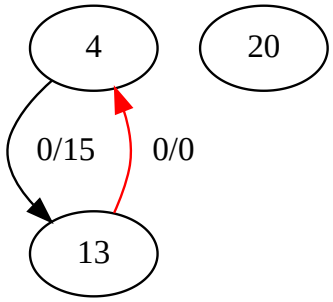
Since this is debug output, IDs are displayed instead of names and residual edges and flows are shown. Here, red edges can be ignored, and you can take the debt along the edge to equal the capacity of the edge.

Though the identifiers are different, the generated graph is in the form of the expected graph

Expected outcome



Generated outcome

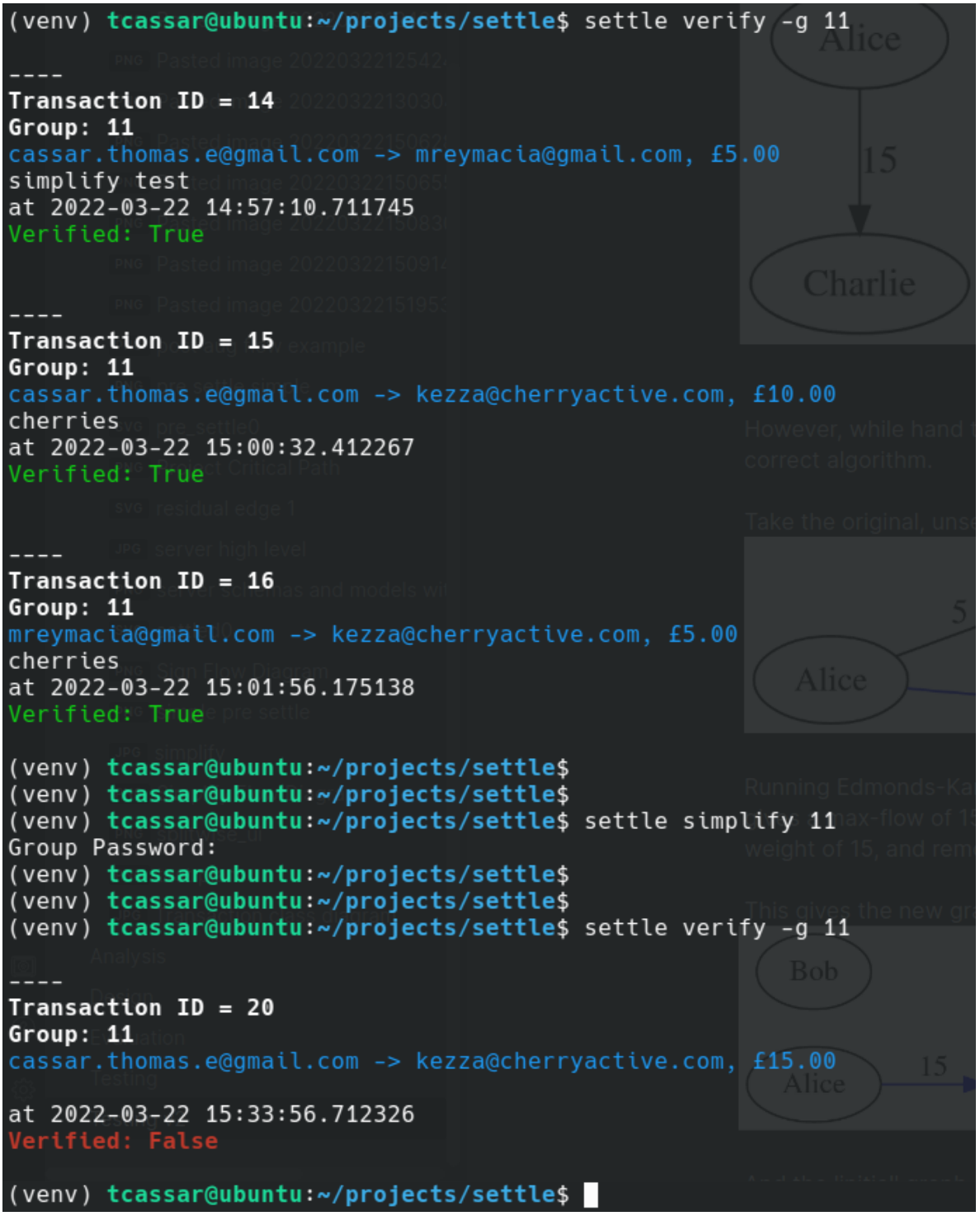


It is clear from these diagrams that the settling process has worked as expected, matching up perfectly with the hand-traced data. The same results are shown in the CLI of the technical solution

The first call `settle verify -g 11` returns all unsettled transactions stored by the database, and checks whether their signatures are valid. In this case, all signatures are valid and thus the group can be simplified. Transactions of the form of the graphs above are found in the group.

The second call is to `settle simplify 11`. This indicates to the server that group 11 should be simplified. The group's password has been entered correctly and thus simplification can occur.

The simplification process will then verify each transaction once more, and build a flow graph representation of the system of debts. My algorithm based on Edmonds-Karp is then run, and a single new transaction is generated. Again, this new transaction is in the form of the graphs above. proving the effectiveness of the solution



The transaction that is generated is marked as unverified. This is because the sever does not have access to the private keys of the users. If the server held user private keys, it would not be a convincing security solution to say the least!

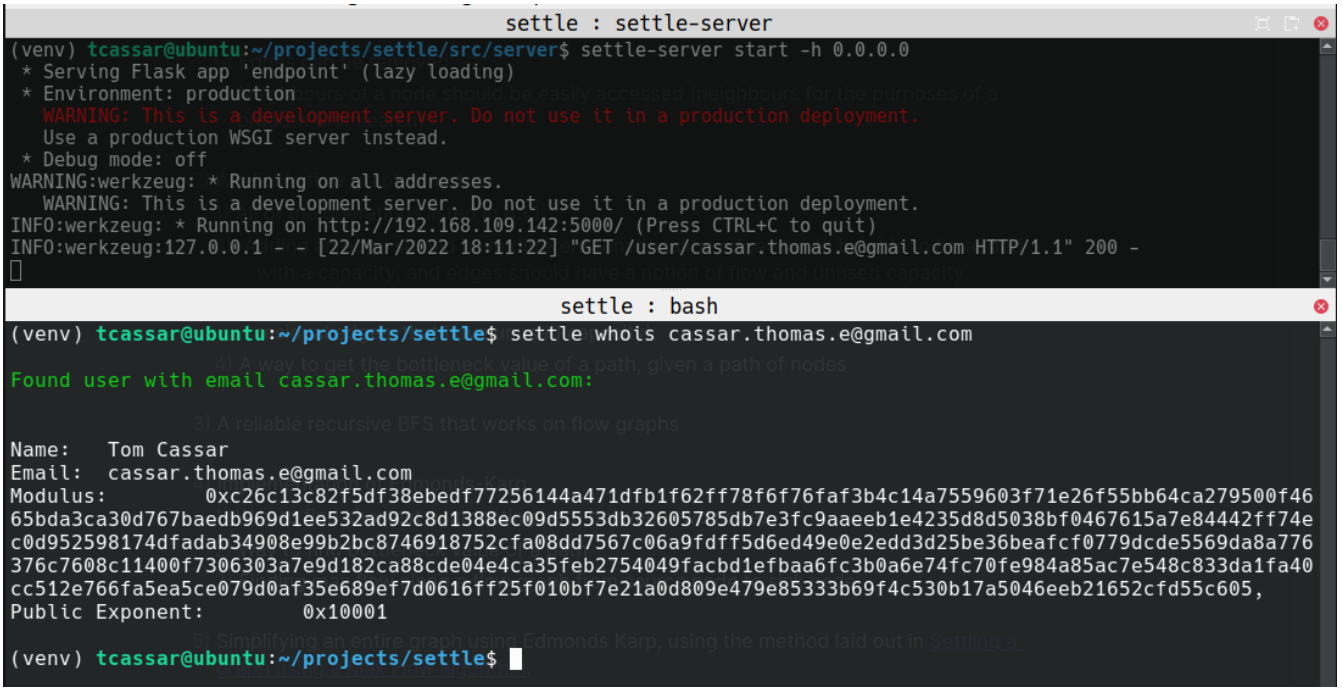
This example fulfils requirements **D1.6**, **D2.1**, **D2.4**, and **D2.5**

A note on UI: the unverified flashes. This is hard to put in a screenshot.

Hence, I can show that every requirement in section A and B has been met.

Evidence of Meeting Requirements - Section C

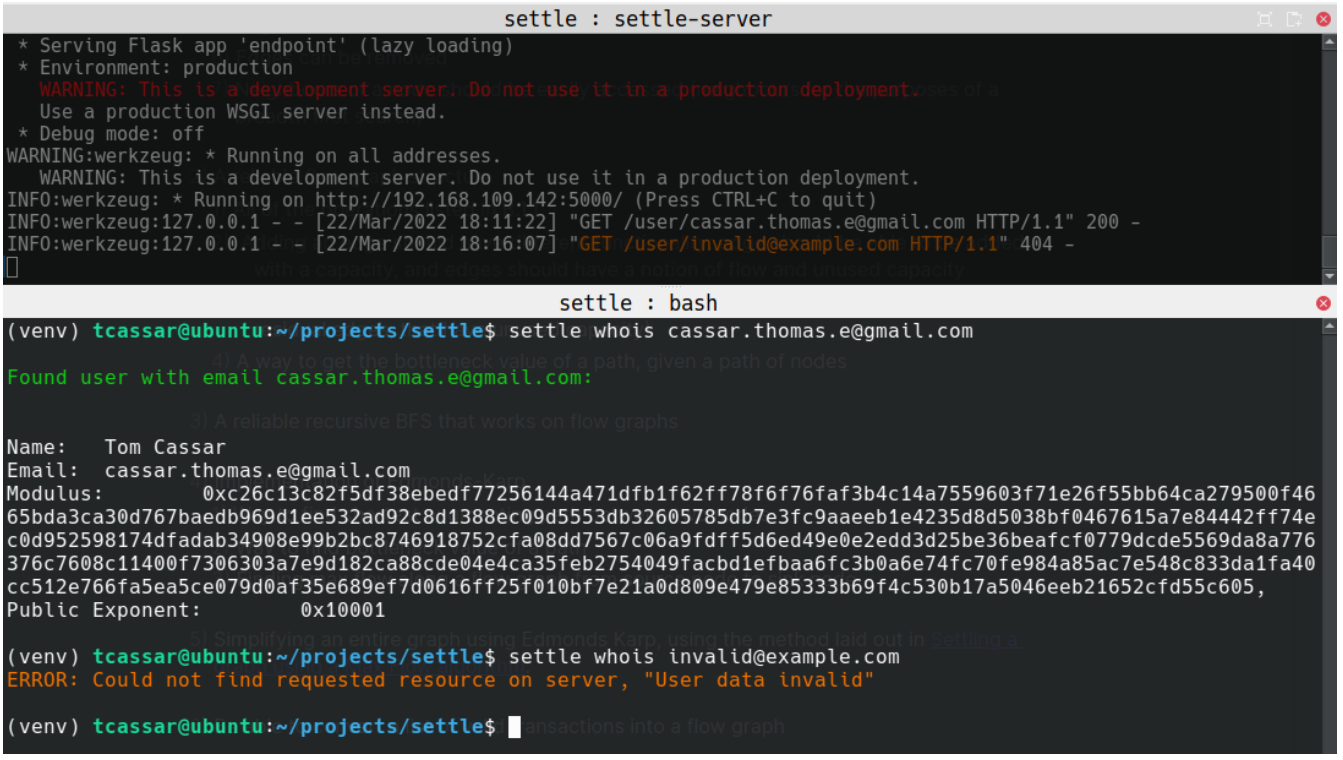
To show that my technical solution does in fact communicate over a REST API, I will show a simple get request made by the client and the corresponding logs produced by the server.



Here, it is possible to see that I have configured the API to run on the `http://192.168.109.142:5000`.

The `settle whois <email>` command returns a user's name and public/private keys given an email. This shows that the server is accessible to the client via a REST api. There is a visible `GET` request, followed by an endpoint of the API. This satisfies **C1**

To show the program fulfilling **C2**, (handling exception codes gracefully), here is what happens when you attempt to `whois` an email that does not exist.

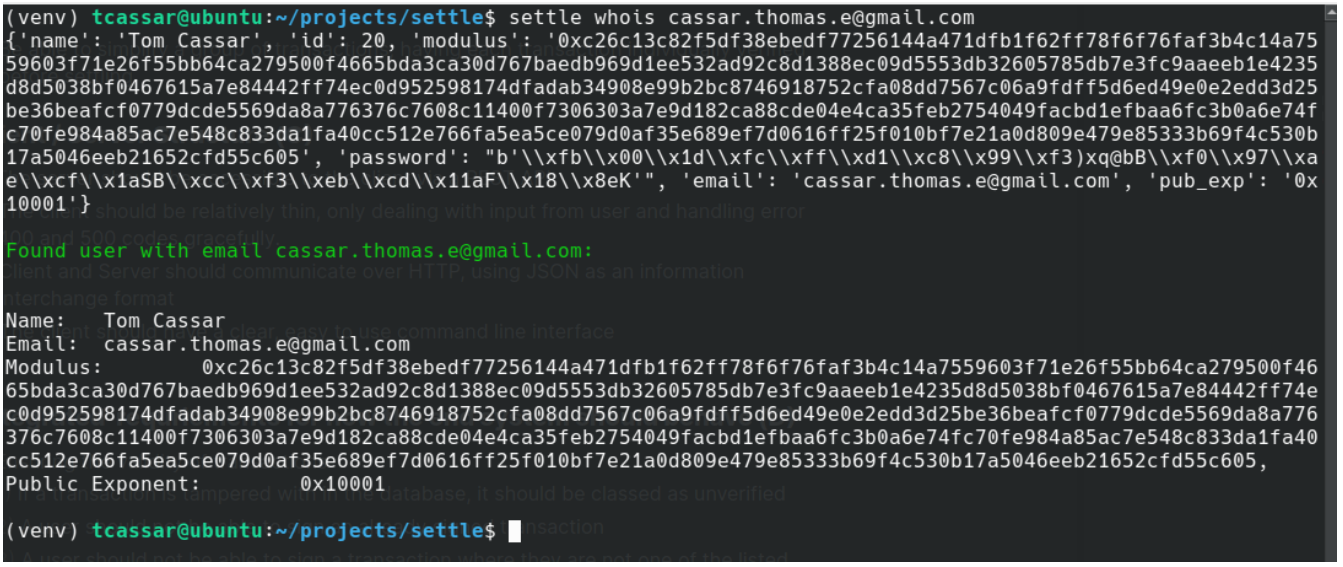


The server has responded with a 404 error code to the client's GET request. On the client side, this has been detected and raised. It has then been handled gracefully before showing the client the type of error (a resource not found), and what that might be (invalid user info).

This type of behaviour is implemented for all errors that are expected to be raised during the programs normal operation. More severe errors, such as the warning when the user attempts to double sign a transaction, are coloured in red.

```
(venv) tcassar@ubuntu:~/projects/settle$ settle sign 12 ./src/crypto/sample_keys/t_private-key.pem
Email: cassar.thomas.e@gmail.com
Password:
        Key validated against server
Authorisation Error; aborting...
Failed to sign transaction: 12
Cannot overwrite signature of src
(venv) tcassar@ubuntu:~/projects/settle$
```

Above, it was shown that the client connects to the API using HTTP. This partially fulfills requirement **C3**. To fulfill it, it must be shown that JSON is used to communicate between client and server. Thus, I have briefly modified the `whois` command to dump the raw server response to STDOUT so that I can prove that this objective has been met (it was reverted after the test)



This clearly shows that JSON is the information interchange format being used.

The screenshots of my CLI I have provided, I believe, effectively demonstrate a clear, easy to use command line interface. Thus, **C4** is also satisfied. Many more examples of output will be shown in the next section.

I can therefore say I have completed all of my requirements in Section C.

Evidence of Meeting Requirements - Section D

Ensuring the validity of transactions

Upon inspecting transaction 13

```
(venv) tcassar@ubuntu:~/projects/settle$ settle verify -t 13

----
Transaction ID = 13
Group: 11
cassar.thomas.e@gmail.com -> mreymacia@gmail.com, £12.87
chickens to simplify
at 2022-03-22 12:41:46.332096
Verified: True

(venv) tcassar@ubuntu:~/projects/settle$
```

Changing the amount owed in the database

id	pair_id	group_id	amount	src_key
2	11	3	1299	
3	23	3	1299	
4	24	3	1000	
5	24	3	1000	
6	24	3	1000	
7	24	3	1500	
8	24	3	1500	
9	32	3	500	
10	24	3	1299	
11	34	3	1387	
12	34	3	1387	
13	34	11	1490	

(1287 → 1490)

Querying again (here, the output of the old query is included first)

```
(venv) tcassar@ubuntu:~/projects/settle$ settle verify -t 13

----
Transaction ID = 13
Group: 11
cassar.thomas.e@gmail.com -> mreymacia@gmail.com, £12.87
chickens to simplify
at 2022-03-22 12:41:46.332096
Verified: True

(venv) tcassar@ubuntu:~/projects/settle$ settle verify -t 13

----
Transaction ID = 13
Group: 11
cassar.thomas.e@gmail.com -> mreymacia@gmail.com, £14.90
chickens to simplify
at 2022-03-22 12:41:46.332096
Verified: False

(venv) tcassar@ubuntu:~/projects/settle$
```

Hence, **D1.1** is shown to be fulfilled.

Note that this transaction is signed. If I re-tamper with it to return the amount to 1287, it becomes valid again. I will show an attempt to re-sign it.

```
(venv) tcassar@ubuntu:~/projects/settle$ settle verify -t 13

----
Transaction ID = 13
Group: 11
cassar.thomas.e@gmail.com -> mreymacia@gmail.com, £12.87
chickens to simplify
at 2022-03-22 12:41:46.332096
Verified: True

(venv) tcassar@ubuntu:~/projects/settle$ settle sign 13
Usage: settle sign [OPTIONS] TRANSACTION_ID KEY_PATH
Try 'settle sign --help' for help.

Error: Missing argument 'KEY_PATH'.
(venv) tcassar@ubuntu:~/projects/settle$ settle sign 13 ./src/crypto/sample_keys/t_private-key.pem
Email: cassar.thomas.e@gmail.com
Password:
        Key validated against server
Authorisation Error; aborting...
Failed to sign transaction: 13
Cannot overwrite signature of src
(venv) tcassar@ubuntu:~/projects/settle$
```

To demonstrate how the CLI handles invalid arguments, I attempted to sign an invalid transaction without a key. The CLI prompted me accordingly, and offered me the choice of using a `--help` flag.

D1.2 is thus fulfilled.

Below, I try to sign a transaction between `cassar.thomas.e@gmail.com` and `kezza@cherryactive.com` as a user on the account held by `mreymacia@gmail.com`. This is not allowed, fulfilling **D1.3**

```
(venv) tcassar@ubuntu:~/projects/settle$ settle verify -t 5

----
Transaction ID = 5
Group: 3
kezza@cherryactive.com -> cassar.thomas.e@gmail.com, £10.00
test
at 2022-03-21 10:44:16.839319
Verified: False

(venv) tcassar@ubuntu:~/projects/settle$ settle sign 5 ./src/crypto/sample_keys/m_private-key.pem
Email: mreymacia@gmail.com
Password:
Email provided doesn't match any of the users in the transaction
(venv) tcassar@ubuntu:~/projects/settle$
```

Next, I try to sign the transaction as `cassar.thomas.e@gmail.com`, but with a different key to the one I registered my account with, fulfilling **D1.4**

```
(venv) tcassar@ubuntu:~/projects/settle$ settle sign 5 ./src/crypto/sample_keys/m_private-key.pem
Email: cassar.thomas.e@gmail.com
Password:
Authorisation Error; aborting...
Private key provided does not match the listing in the db
(venv) tcassar@ubuntu:~/projects/settle$
```

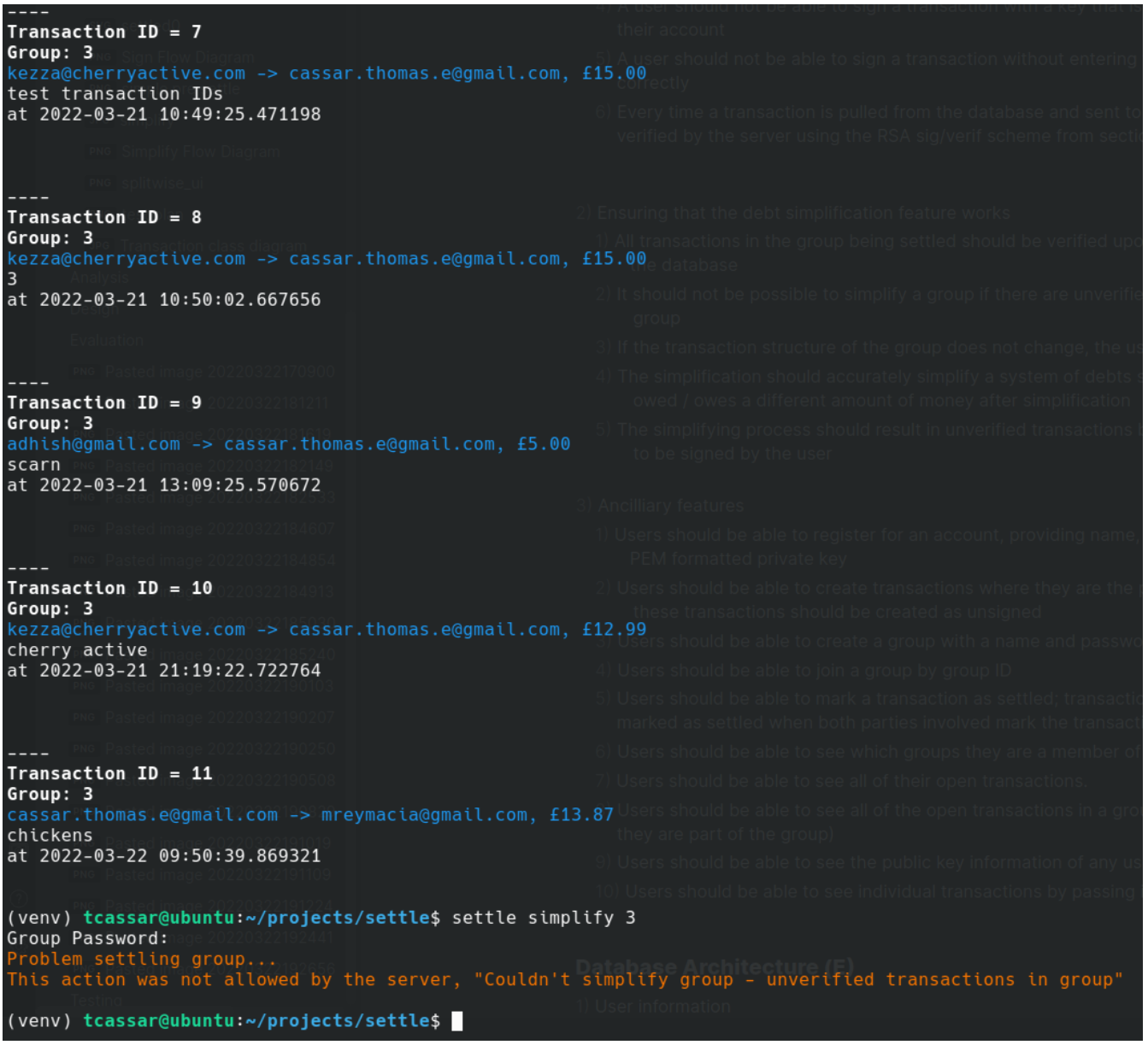
I attempt to sign the transaction with my private key, but I mistype my password **D1.5**

```
(venv) tcassar@ubuntu:~/projects/settle$ settle sign 5 ./src/crypto/sample_keys/t_private-key.pem
Email: cassar.thomas.e@gmail.com
Password:
Authorisation Error; aborting...
Password Incorrect
(venv) tcassar@ubuntu:~/projects/settle$
```

D1.6 was shown previously.

D2.1, **D2.4** and **D2.5** were shown previously.

To show that I have fulfilled **D2.2**, I will attempt to settle a group with unverified transactions



As aforementioned, the `Verified: False` messages blink. This is why they are not shown in this screenshot (evidence of blinking in video provided).

To show that I satisfy **D2.3**, **D3.2**, **D3.3**, **D3.4**, and **D3.10** I will make a new group and add two users to it. I will then add a single transaction and attempt to settle. One transaction cannot be simplified, thus we should be warned that simplification cannot happen

```
(venv) tcassar@ubuntu:~/projects/settle$ settle new-group
Name: test d1.7
Password:
Repeat for confirmation:
"Created group ID=12 named test d1.7"

You can join this group with `settle join`
(venv) tcassar@ubuntu:~/projects/settle$ settle join 12
Email: cassar.thomas.e@gmail.com
Your password:
Group Password:
Successfully joined group 12
(venv) tcassar@ubuntu:~/projects/settle$ settle join 12
Email: kezza@cherryactive.com
Your password:
Group Password:
Authorisation Error; aborting...
Password Incorrect
(venv) tcassar@ubuntu:~/projects/settle$ settle join 12
Email: kezza@cherryactive.com
Your password:
Group Password:
Successfully joined group 12
```

D3.3 and **D3.4** have been fulfilled

Making a transaction (thus fulfilling **D3.2** and **D3.10**)

```
(venv) tcassar@ubuntu:~/projects/settle$ settle new-transaction
Email of payee: kezza@cherryactive.com
Amount (in GBP): 28
Reference: test one transaction in a group
Group: 12
Your email: cassar.thomas.e@gmail.com
Password:
Transaction generated with ID=24
Sign with `settle sign 24`
(venv) tcassar@ubuntu:~/projects/settle$
```

It is initially unsigned - I then sign it with both users

```
(venv) tcassar@ubuntu:~/projects/settle$ settle verify -t 24

----
Transaction ID = 24
Group: 12
cassar.thomas.e@gmail.com -> kezza@cherryactive.com, £28.00
test one transaction in a group
at 2022-03-22 19:29:48.330112
Verified: False

(venv) tcassar@ubuntu:~/projects/settle$ settle sign 24 ./src/crypto/sample_keys/d_private-key.pem
Email: kezza@cherryactive.com
Password:
        Key validated against server
        successfully signed transaction
        Successfully appended signature in database!
(venv) tcassar@ubuntu:~/projects/settle$ settle sign 24 ./src/crypto/sample_keys/t_private-key.pem
Email: cassar.thomas.e@gmail.com
Password:
        Key validated against server
        successfully signed transaction
        Successfully appended signature in database!
(venv) tcassar@ubuntu:~/projects/settle$ settle verify -t 24

----
Transaction ID = 24
Group: 12
cassar.thomas.e@gmail.com -> kezza@cherryactive.com, £28.00
test one transaction in a group
at 2022-03-22 19:29:48.330112
Verified: True

(venv) tcassar@ubuntu:~/projects/settle$
```

It is now verified. Next, I try to settle group 12. Since it only one transaction, I should be alerted that no changes were made.

```
(venv) tcassar@ubuntu:~/projects/settle$ settle simplify 12
Group Password:
No changes were made
"No changes made to debt structure - heuristic did not find anywhere to simplify"

(venv) tcassar@ubuntu:~/projects/settle$
```

Indeed, I am.

To show **D3.1**, I will register an account, first providing it with a public key. It should reject this. Then, I will then register the account with the correct key. I will confirm the creation of the account with the `settle whois` command, thus also fulfilling **D3.9**

```
(venv) tcassar@ubuntu:~/projects/settle$ settle register
Full Name: example person
Email: example@gmail.com
Password:
Repeat for confirmation:
Path to RSA key: /home/tcassar/projects/settle/src/crypto/sample_keys/d_public-key.pe
unable to load Private Key
139870883080000:error:0909006C:PEM routines:get_name:no start line:../crypto/pem/pem_lib.c:745:Expecting: ANY PRIVATE KEY
Failed to create account - issue with given RSA key;
File not in PEM private key format
(venv) tcassar@ubuntu:~/projects/settle$
```

Successful creation

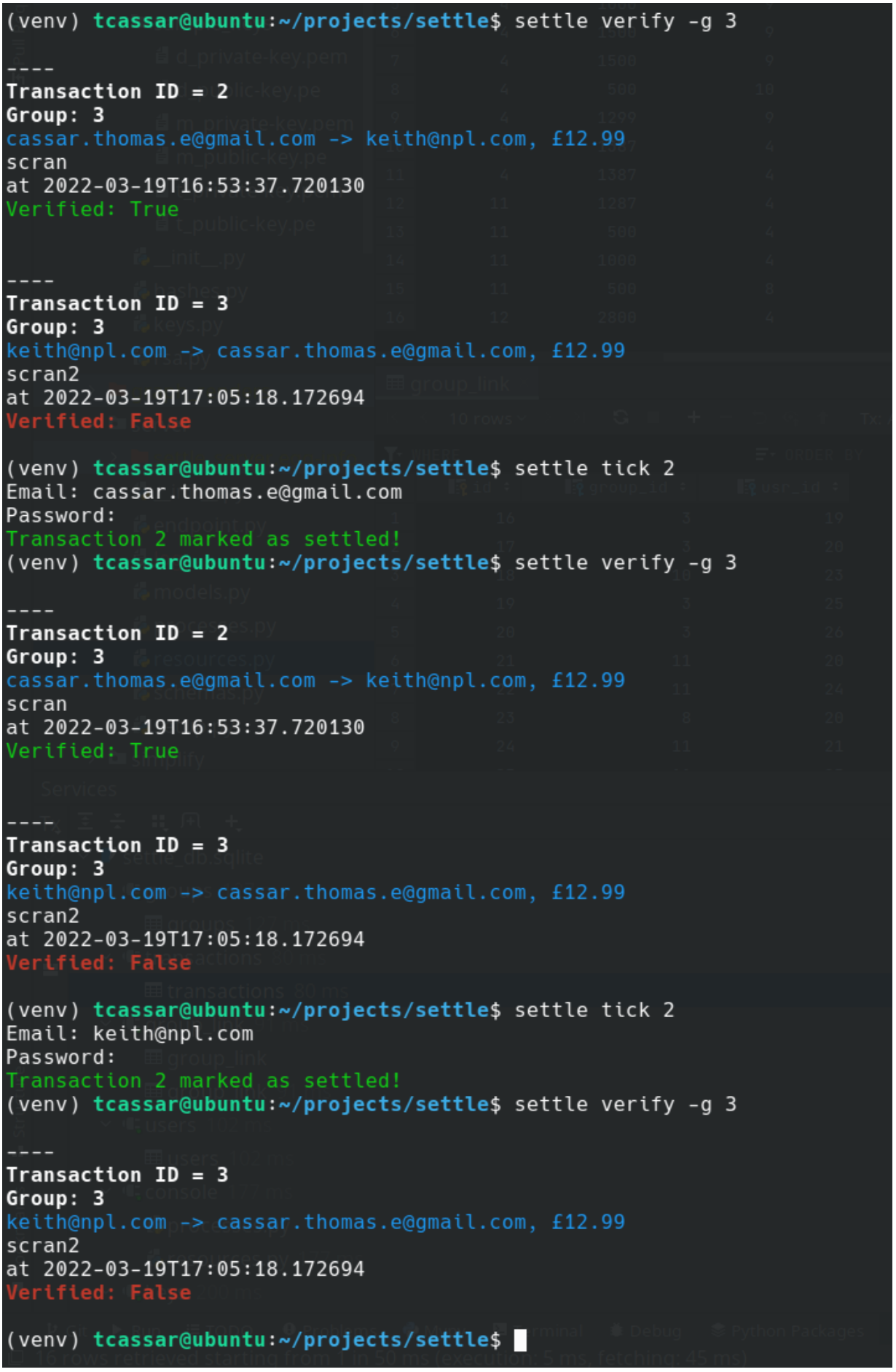
```
(venv) tcassar@ubuntu:~/projects/settle$ settle register
Full Name: example person
Email: example@gmail.com
Password:
Repeat for confirmation:
Path to RSA key: /home/tcassar/projects/settle/src/crypto/sample_keys/t_private-key.pem
Account created successfully

Name:   example person
Email:  example@gmail.com
Modulus:        0xc26c13c82f5df38ebedf77256144a471dfb1f62ff78f6f76faf3b4c14a7559603f71e26f55bb64ca279500f4665bda3ca30d767baed
b969d1ee532ad92c8d1388ec09d5553db32605785db7e3fc9aaeeb1e4235d8d5038bf0467615a7e84442ff74ec0d952598174dfadab34908e99b2bc874691
8752cfa08dd7567c06a9fdff5d6ed49e0e2edd3d25be36beafcf0779dcde5569da8a776376c7608c11400f7306303a7e9d182ca88cde04e4ca35feb275404
9facbd1efbaa6fc3b0a6e74fc70fe984a85ac7e548c833da1fa40cc512e766fa5ea5ce079d0af35e689ef7d0616ff25f010bf7e21a0d809e479e85333b69f
4c530b17a5046eeb21652cfd55c605,
Public Exponent:        0x10001

(venv) tcassar@ubuntu:~/projects/settle$ settle whois example@gmail.com

Found user with email example@gmail.com:


Name:   example person
Email:  example@gmail.com
Modulus:        0xc26c13c82f5df38ebedf77256144a471dfb1f62ff78f6f76faf3b4c14a7559603f71e26f55bb64ca279500f4665bda3ca30d767baed
b969d1ee532ad92c8d1388ec09d5553db32605785db7e3fc9aaeeb1e4235d8d5038bf0467615a7e84442ff74ec0d952598174dfadab34908e99b2bc874691
8752cfa08dd7567c06a9fdff5d6ed49e0e2edd3d25be36beafcf0779dcde5569da8a776376c7608c11400f7306303a7e9d182ca88cde04e4ca35feb275404
9facbd1efbaa6fc3b0a6e74fc70fe984a85ac7e548c833da1fa40cc512e766fa5ea5ce079d0af35e689ef7d0616ff25f010bf7e21a0d809e479e85333b69f
4c530b17a5046eeb21652cfd55c605,
Public Exponent:        0x10001

(venv) tcassar@ubuntu:~/projects/settle$
```

To show fulfillment of **D3.5**, **D3.7**, and **D3.8**. I will find an open transaction and mark it as settled. It should no longer appear in a set of group debts after both parties mark it as settled, but should appear if only one party has marked it as settled.

```
(venv) tcassar@ubuntu:~/projects/settle$ settle show -t
Email: keith@npl.com

You owe cassar.thomas.e@gmail.com £12.99

Reference: scran2
Agreed upon at 2022-03-19T17:05:18.172694ID: 3
Unverified

cassar.thomas.e@gmail.com owes you £12.99

Reference: scran
Agreed upon at 2022-03-19T16:53:37.720130ID: 2
Unverified
----------

You owe and are owed nothing; all debts settled
Your unverified totals => all debts settled
(venv) tcassar@ubuntu:~/projects/settle$
```

I will now sign transaction 2 by both parties - this has already been documented and thus won't be shown again.

Showing group three (and fulfilling **D3.8**), we see two transactions. After one party marks it as settled, it is still classed as an open transaction. Once the second party has ticked it off it is counted as settled. Hence, it is no longer shown with the rest of the open group transactions.



This shows transactions 2 and 3 listed as unmarked until both parties have ticked transaction 2. In the final call to `verify`, only transaction 3 is shown. Hence, **D3.5** and **D3.7** have been met.

Evidence for **D3.6**:

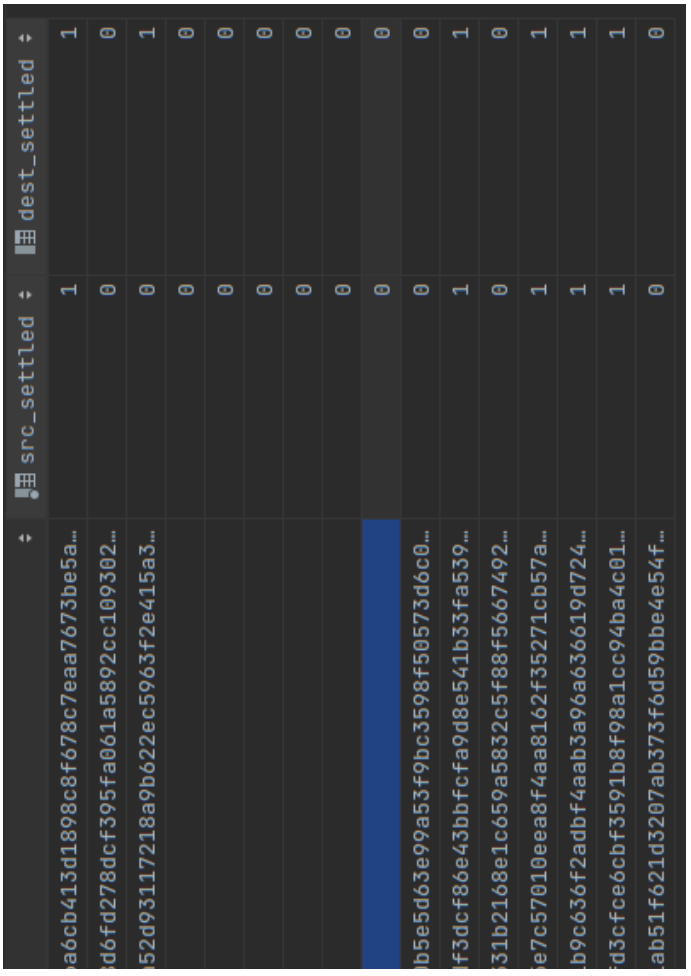
```
(venv) tcassar@ubuntu:~/projects/settle$ settle show -g
Email: cassar.thomas.e@gmail.com
Groups that you are a member of:

Group:
        Name: test group, ID = 3
Group:
        Name: simplify test, ID = 11
Group:
        Name: test, ID = 8
Group:
        Name: test d1.7, ID = 12

(venv) tcassar@ubuntu:~/projects/settle$
```

Evidence of Meeting Requirements - Section E

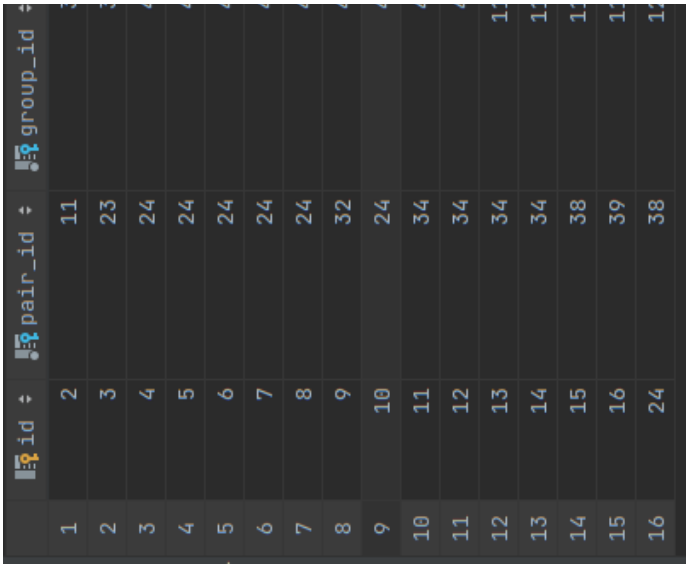
To show that I have implemented the database structure that I laid out in my requirements, I will screenshot the database tables, and briefly comment on which requirements each table fulfils.

Not only does this show the structure of the data being stored, it also proves that I am storing data as per my requirements.

Transactions
This table fulfills **E2.1 → E2.8** in conjunction with the `pairs` table and the `keys` table. It also satisfies **E3.4**



	amount	src_key	dest_key	reference	time_of_creation	src_sig	dest_sig
5	1299	4	5	scran	2022-03-19T16:53:37.720130	0x6619229faa043a839cc3595bf1284da28b58aa95b4e…	0xae6de6b18256
5	1299	5	4	scran2	2022-03-19T17:05:18.172694		0x55abc6445a38
4	1000	9	4	security transaction	2022-03-21 10:05:48.828933		0x9e0d44e0e8fa
4	1000	9	4	test	2022-03-21 10:44:16.839319		
4	1000	9	4	kez money	2022-03-21 10:47:25.463493		
4	1500	9	4	test transaction IDs	2022-03-21 10:49:25.471198		
4	1500	9	4	3	2022-03-21 10:50:02.667656		
4	500	10	4	scarn	2022-03-21 13:09:25.570672		
4	1299	9	4	cherry active	2022-03-21 21:19:22.722764		
4	1387	4	8	chickens	2022-03-22 09:50:39.869321		0x8f57244be550
4	1387	4	8	chickens	2022-03-22 09:55:42.800220	0x30a31c8375275d778542c9332f56e0232b83fc22750…	0xb1da9bcfe00d
1	1287	4	8	chickens to simplify	2022-03-22 12:41:46.332096	0x839075842746e0cb35b010118478dffccfc670311c6…	0x5f7fbabdf263
1	500	4	8	simplify test	2022-03-22 14:57:10.711745	0x249baaf2517307987f154619a99f136109357b285d6…	0x75d4f3e11805
1	1000	4	9	cherries	2022-03-22 15:00:32.412267	0x4bf53cfa1ab337b59dbf69e82905841f891a4c5dad4…	0x7f68499aed51
1	500	8	9	cherries	2022-03-22 15:01:56.175138	0x1fda9bf10034dad2689f4285b9af7de75adfa69616b…	0x77decc27c32f
2	2800	4	9	test one transactio…	2022-03-22 19:29:48.330112	0xaaa65d3ddd7dbd1baa1da6d74d6c87b778a59fe2354…	0x71c2db2484b1



Pairs

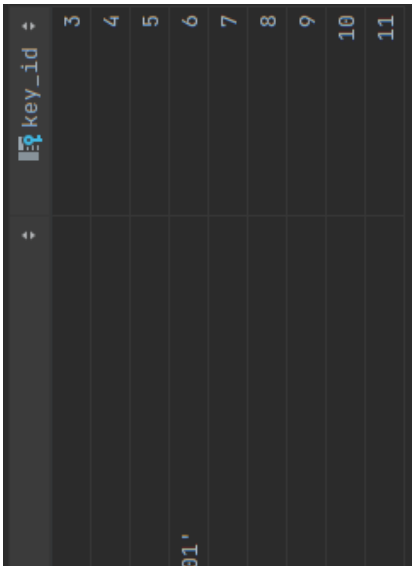
	id	src_id	dest_id
1	11	20	21
2	23	21	20
3	24	25	20
4	32	26	20
5	34	20	24
6	38	20	25
7	39	24	25

Keys

	id	n	e
1	3	0xc26c13c82f5df38ebedf77256144a471dfb1f62ff78f6f76faf3b4c14a7559603f71e26f55bb64ca279500f4665bda3ca30…	0x10001
2	4	0xc26c13c82f5df38ebedf77256144a471dfb1f62ff78f6f76faf3b4c14a7559603f71e26f55bb64ca279500f4665bda3ca30…	0x10001
3	5	0xb2c18e327280ff4abe3ed337b022518563a336871851178f9ee13cd8085c875c080184d836ca0a3f2d1fb771c98931cdbe2…	0x10001
4	6	0xb2c18e327280ff4abe3ed337b022518563a336871851178f9ee13cd8085c875c080184d836ca0a3f2d1fb771c98931cdbe2…	0x10001
5	7	0xc26c13c82f5df38ebedf77256144a471dfb1f62ff78f6f76faf3b4c14a7559603f71e26f55bb64ca279500f4665bda3ca30…	0x10001
6	8	0xb2c18e327280ff4abe3ed337b022518563a336871851178f9ee13cd8085c875c080184d836ca0a3f2d1fb771c98931cdbe2…	0x10001
7	9	0xab3cec18ae8df68c502ed71ce3376cca00c5e82aeebf9aa504794ed2b4b340093e381cbe2dc84c274da640a2d2c379a8c47…	0x10001
8	10	0xc26c13c82f5df38ebedf77256144a471dfb1f62ff78f6f76faf3b4c14a7559603f71e26f55bb64ca279500f4665bda3ca30…	0x10001
9	11	0xc26c13c82f5df38ebedf77256144a471dfb1f62ff78f6f76faf3b4c14a7559603f71e26f55bb64ca279500f4665bda3ca30…	0x10001

Users

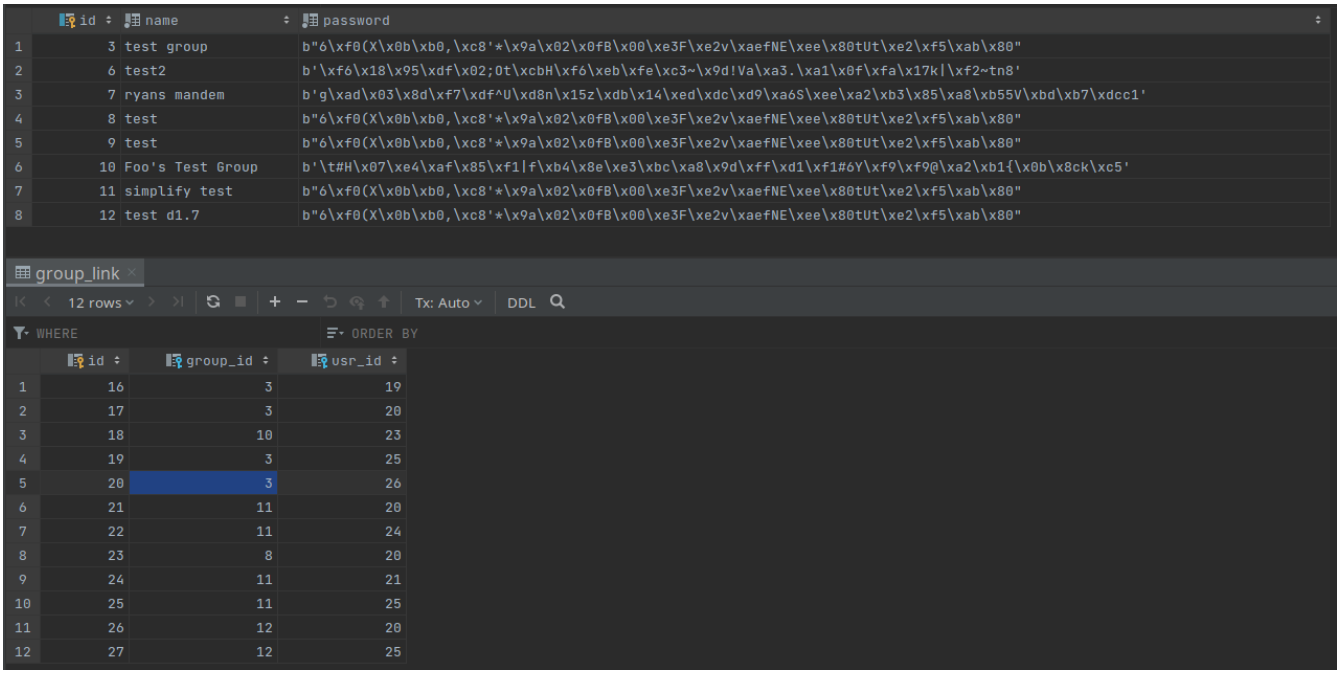
The users table, along with the keys table above, fulfil requirements **E1.1 → E1.5**



name	email	password
dmin	admin@admin.com	b'\xfb\x00\x1d\xfc\xff\xd1\xc8\x99\xf3)xq@bB\xf0\x97\xae\xcf\x1aSB\xcc\xf3\xeb\xcd\x11aF\x18\x8eK'
om Cassar	cassar.thomas.e@gmail.com	b'\xfb\x00\x1d\xfc\xff\xd1\xc8\x99\xf3)xq@bB\xf0\x97\xae\xcf\x1aSB\xcc\xf3\xeb\xcd\x11aF\x18\x8eK'
unes Tahir	keith@npl.com	b'\x90\xd9D\x8a6U\xc6<\x04\x92\x14Z\x86\xbd\xecl\xb2\x01K-H\xacTB\xdeD=M\xab\xd1\xf2\xf8'
oo	foo@bar.com	b'v\xd3\xbcA\xc9\xf5\x88\xf7\xfc\xd0\xd5\xbfG\x18\xf8\xf8K\x1cA\xb2\x08\x82p1\x00\xb9\xeb\x94\x13\x80|\x0
oo Bar	foobar@example.com	b'\t#H\x07\xe4\xaf\x85\xf1|f\xb4\x8e\xe3\xbc\xa8\x9d\xff\xd1\xf1#6Y\xf9\xf9@\xa2\xb1{\x0b\x8ck\xc5'
aia Rey Macia	mreymacia@gmail.com	b'8\xdd\x08\x8b5S\x81>\x8e0p/\xf6\x16o(\x97^\xc6E\xb0AN\xcdj\x89\n/0\x87^l'
ez Carey	kezza@cherryactive.com	b'\xff\x8es\xe7\xb3\x1f\x12\x1e\xeb\xbe46wZ\x81;\xf4`\xbf\xd1\xd8\xd0\xe4\x9dH]k\n\x14L\xcf~'
dhish	adhish@gmail.com	b'\xf5\x1a\xfe\x17\xc9\xe1\x85\xc9\xce\xb9\x9aoV\xd1/\x19K\xec\xe2m\x19c\xb9\tY\x86\x8eJ\x9c\xf4\x9bp'
xample person	example@gmail.com	b'p\x98=i/d\x81\x85\xfe\xbemo\xa6\x07c\n\xe6\x86I\xf7\xe6\xfcE\xb9F\x80\tl\x06\xe4\xfa\xdb'

Groups

The final outstanding requirements in Section E are met by the `groups` table and `groups_link` table, meeting **E3.1 → E3.3**



	id	name	password
1	3	test group	b"6\xf0(X\x0b\xb0,\xc8'*\x9a\x02\x0fB\x00\xe3F\xe2v\xaefNE\xee\x80tUt\xe2\xf5\xab\x80"
2	6	test2	b'\xf6\x18\x95\xdf\x02;Ot\xcbH\xf6\xeb\xfe\xc3~\x9d!Va\xa3.\xa1\x0f\xfa\x17k|\xf2~tn8'
3	7	ryans mandem	b'g\xad\x03\x8d\xf7\xdf^U\xd8n\x15z\xdb\x14\xed\xdc\xd9\xa6S\xee\xa2\xb3\x85\xa8\xb55V\xbd\xb7\xdcc1'
4	8	test	b"6\xf0(X\x0b\xb0,\xc8'*\x9a\x02\x0fB\x00\xe3F\xe2v\xaefNE\xee\x80tUt\xe2\xf5\xab\x80"
5	9	test	b"6\xf0(X\x0b\xb0,\xc8'*\x9a\x02\x0fB\x00\xe3F\xe2v\xaefNE\xee\x80tUt\xe2\xf5\xab\x80"
6	10	Foo's Test Group	b'\t#H\x07\xe4\xaf\x85\xf1|f\xb4\x8e\xe3\xbc\xa8\x9d\xff\xd1\xf1#6Y\xf9\xf9@\xa2\xb1{\x0b\x8ck\xc5'
7	11	simplify test	b"6\xf0(X\x0b\xb0,\xc8'*\x9a\x02\x0fB\x00\xe3F\xe2v\xaefNE\xee\x80tUt\xe2\xf5\xab\x80"
8	12	test d1.7	b"6\xf0(X\x0b\xb0,\xc8'*\x9a\x02\x0fB\x00\xe3F\xe2v\xaefNE\xee\x80tUt\xe2\xf5\xab\x80"

group_link

12 rows · Tx: Auto · DDL

WHERE ORDER BY

	id	group_id	usr_id
1	16	3	19
2	17	3	20
3	18	10	23
4	19	3	25
5	20	3	26
6	21	11	20
7	22	11	24
8	23	8	20
9	24	11	21
10	25	11	25
11	26	12	20
12	27	12	25

Hence, I have entirely fulfilled every requirement I outlined in Section E, and have thus completed my project entirely.

Evaluation

I will evaluate my completed project by gaining the opinion of the end user that I talked in the Analysis phase. I plan to give him a demonstration of the project, and let him use the project for a week. I will then collect and reflect on his feedback.

In the demo, I thought it to be important to directly address the main concerns that he originally highlighted. These were mainly centred around system administrators (me) altering the amount of money that people owe each other.

To put his mind at ease, I gave him a live demonstration of the tampering test I used to show that my system fulfilled requirement **D1.1**. He was extremely impressed with this and spent the next 5 minutes tampering with data in the database, and watching the previously verified transactions become unverified.

Once he had convinced himself that public key cryptography works, we moved on to addressing his next concern - debt simplification. Again, I showed him the 'simplify debt' example that I used in my analysis, and we spent the next 10 minutes building graphs and watching them simplify down.

During this, he became increasingly comfortable using the CLI. He told me he was a bit hesitant when he saw it - it looked like nothing he had ever used before. While playing with the security and debt simplification features of the app, it was quite interesting to see just how quickly he got used to it. He told me that a CLI was definitely a good choice for this, due to just how simple it is to get things done.

However, he was not completely without criticism.

He reported, fairly, that it was slightly annoying to have to keep entering your email and password, especially when he had entered this information just one command before.

He also said that he would like a way to see his closed transactions in a list view, not just accessing them by ID.

Finally, he said that he was confused as to why he had to generate an OpenSSL RSA Private key himself and then feed it into the `settle register` command. In his opinion, a key generate command would have been helpful.

I think that these are entirely valid concerns, and would be where I go next if I were to continue to improve the project.

I decided to look into how I would go about implementing the end user's suggestions.

The email / password remembering could be achieved relatively straightforwardly with the `click` library using the concept of `contexts`. This is most definitely something that is doable, and would do a lot to aid the overall user experience.

Similarly, it would be trivial to add use a binding to OpenSSL and have my client be able to generate an RSA private key through the CLI. This is just something I hadn't considered in my Analysis of the project, but would definitely add to the user experience.

Adding a view for closed transactions would be another command a modified SQL statement, and I would probably need to add query parsing to my API endpoints, so that I could keep my server processes almost identical (aside from writing the new statement and query parser). I would then be able to pull transactions either open or closed, fully with code that already exists.

However, I was on the whole extremely happy with his response. When I asked him on how good a fit my solution to his problem was, he told me that with his additions, it would be absolutely perfect.

When considering the extent to which I met my own high level requirements, I am really quite pleased. While everything was planned and meticulously designed, I am still surprised at just how robust my end product is.

Individual Requirement Reflection

An RSA implementation that will allow the signing, and verification, of transactions

On the whole I have built a robust, functional implementation of RSA for this project, which effectively fulfils the aim of preventing tampering with transactions (as well as more menial things such as password hashing). If I were to do this project again, however, I would not have written the cryptography side of things in Python.

Due to the way that Python stores numbers, it is impossible to rule out the possibility of side channel attacks. Similarly, I did not use a constant time algorithm for the modular exponentiation step of RSA. Thus, it would be possible for an attacker to use a timing attack to deduce the private exponent in a user's private key.

I think in the context that I had intended the project to be used in, this is not currently a massively pressing issue. Knowing about it does mean, however, that I would like to protect against it even if it means learning a new programming language.

A way to settle the debts of the group in as few as possible (heuristically speaking) monetary transfers

I am most pleased with the debt simplification - I think that that is a genuinely useful idea, and I am happy that it works so well. I did learn some interesting things about how the heuristic model behaved when I was experimenting with making graphs with my end user.

I discovered that the algorithm that I implemented to simplify groups of debt works best on densely connected graphs as more augmenting paths can be found. This makes me wonder if there is a way to change how I search for augmenting paths in the flow graph to try and end up with longer chains of debt.

The problem is that a path of $n + 1$ edges has a higher chance of a smaller bottleneck value than a path with n edges, and could end up leading to the time complexity of the algorithm being dependent on flow. This would potentially make the algorithm less efficient.

This is something that I would like to investigate more in the future.

Another thing that I would like to test is adding cycle detection to the graph. I feel as though I could improve the performance of my heuristic if I simplified cycles before running any flow graph algorithms. However, this may worsen the time complexity of the process as a whole. This, it is another thing to experiment with.

A server-side component of the application which can verify transactions, and store / retrieve them from a database

A client-side component of the application that will have a simple user interface (CLI)

A screenshot of the simple, easy to use CLI

The next two requirements can be addressed as one. I am very happy with the client-server model of the system, even though the client-server model was much more work than I imagined. However, it is quite impressive to see communication across a network, even if it is currently a localhost network.

Having a thin client is particularly easy and effective as it means that, if more people were to adopt this solution and I were to set up a full-time server, this could be run on absolutely everything.

As was discussed with the end user, there are certain improvements that I could make to the CLI. These are very much quality of life improvements, and the CLI that I provided was more than adequate at fulfilling all of my initial requirements

A database that should be able to store user and transaction information

Finally, I am happy with my database. However, as I became more comfortable with SQL during the project, having had absolutely no experience with it before, I realise that I have a redundant relationship in my database design. The transaction table references the keys table. Having learned about the join statement, I now see that this link is redundant and ought not to be there.