Identity layer for SimpleX Chat

Motivation

While lack of global identity provides high level of meta-data privacy and there are communication scenarios where not having to use public identity is desirable (e.g., a customer asking a question in online store before making a purchase), many other communication scenarios require identity (e.g., online store itself would benefit from having a public address via which it can be connected to).

We have already added user contact addresses represented as links/QR codes, but they are impossible to memorize or to say in the conversation - they require digital transmission, while in many scenarios analogue transmission would be more convenient.

There is already a global addressing system in use - email addresses. Email is the only communication system in existence that is fully decentralized and has a quality called "address portability" - that is, users can keep an address while changing the server provider. It is not true in 100% of cases - e.g. if you have an address in gmail.com domain, you cannot move it to another provider. But if you use Google Mail to host your email address in another domain, e.g. as I do with my evgeny@poberezkin.com address, you can move it to another provider without any restriction - it only requires a simple change in the domain DNS record.

Email identities have multiple problems though:

- your providers do not only know that you have an identity on the network, they also know everybody who sends you messages.
- you cannot change your address without losing your contacts - you have to tell all your contacts your new address.
- providers cannot guarantee, in most cases, that the sender of the email has the address they claim they have. At best, they can check the sending domain using DKIM and SPKIF, but these measures are not enabled by default, require complex additional configuration and most small users do not configure it correctly.

The problems with our current user contact address:

- it allows to send traffic directly to user's device (via SMP server), there is no proxy to implement anti-spam/anti-DoS protection. While we can mitigate it by throttling the reception of contact requests on the client side, the attacker can keep the queue full and make the address unreachable. And if we don't throttle, the client can receive unlimited amount of traffic via contact requests and run out of disk space. Having identity server as a proxy allows to manage this problem.

The identity layer we add to SimpleX Chat has to keep the advantages of email addressing and to solve these problems as well.

Product requirements

- the identity of the user in SimpleX Chat must be the same as the email address they have access to - the identity server should prove it while provisioning the address.
- the email access should not be used to access SimpleX address once it's provisioned.
- the identity server and network observers should not know who sends messages to the user's identity - sender's addresses should be only visible to the recipient.
- changing or deleting address/identity should not make you losing the contacts - we already have this quality with our user contact addresses.
- for every email address user possesses, they should be able to create multiple SimpleX addresses that can be used in different contexts using SMTP + extension syntax (TBC what it is called). This extension in case of SimpleX should be meaningful - that is if a given address does not exist, the address should not be reachable.
- Possibly, the user should be able to decide whether the server should screen for known + addresses or if it should be screened on the device, where the server only checks the main part of the address or uses some patterns for + part. It's a trade-off - screening on the device hides the actual addresses from the identity server, but increases the amount of spam traffic the device has to process, while screening on the server reduces the spam traffic, but reduces user privacy, as the server knows all the sub-addresses. From the UX point of view there would be no difference - the user will not see non-existing + extensions, unless they choose to receive connection requests to their main address (that could be put in a separate folder).
- The user should be able to decide whether they want to receive connection requests from users without verified SimpleX addresses (all addresses are verified).
- The senders' addresses should be cryptographically verified by delivering some probe to that address and receiving the reply, transparently to the user, before the connection request is shown to the user.
- Identity server should be accessed via chat clients, it should not have its own separate UI.
- The client of the address owner should be able to periodically confirm that the identity server does not perform MITM attack on the address to see the senders (by substituting the owner's public key for that address).

Post MVP:

- we should support identity transfer, e.g. if email changes the owner.
- we should have some cool-down period for identity transfer, to the reduce risks of malicious address takeover.

Design considerations

- Identity server should communicate with the clients via SMP protocol

- It probably should use the same message format as we use for chat, but in a different namespace (e.g., `i` - we currently use `x` for chat messages).
- Having received the request to provision the address, it should send an email to this address to verify it, following the best anti-SPAM practices and allowing recipients to opt out of further emails.
- Client should determine which SimpleX identity server to use via some DNS record we decide on - we should support TXT and see if we can register something in IANA for DNS servers that support custom records.
- In case domain does not have registered identity server, client could default to hard-coded identity server we provide, that would have a list of domains we allow creating addresses in.

Preliminary protocol design

It is assumed that identity server has it's own contact address, with the same syntax as user's contact address, but with a different mode, e.g. `identity`. Or, possibly, we should use the same `contact` namespace. TBC

Establishing permanent connections with identity server

This connection is only needed for requesting and managing provisioned addresses - this connection identifies the user to the server. As some actions must be anonymous, they should not be sent via this connection.

To provision the connection the client would send `INV` agent message, same as when connecting with other users, but without any user profile.

Requesting the address

1. Client sends `i.addr.new <email> <contact_connection_request> <public_signature_key>` message to identity server via established permanent connection (`contact_connection_request` will be used by the server to forward connection requests to, `public_signature_key` would be used to confirm the identity to the server on all operations with the address. Possibly we don't need it and can rely on SMP authorization/connections, TBC). `email` can include `+` extension or be catch all.
2. Identity server responds with `i.addr.id <address_id>` message to confirm that the request is received and being processed.
3. Unless the main address was already confirmed (e.g. when the client requests additional `+` extension to the previously created address), the identity server will send email to this address with the unique connection request uri that the user should use to confirm this address. Possibly it should have separate mode, e.g. `confirm_address` TBC.
4. Once the user confirms or rejects the address.
    - If address is confirmed (see below), the server will:
        - create new SMP queue(s) for the address that it would process.

        - send `i.addr.conf <address_id> <connection_request_uri>` message to confirm that this identity is now associated with the user (including the created queue(s)). Public key in `connection_request_uri` must be the same as the one the user passed to the server, if it is different it means server does not function correctly and this address should not be used.
    - if the address is rejected, the server will send `i.addr.rejected <address_id> <error_code>`. Servers can have some policies to allow limited number of rejections per confirmed connection, and also block creating the new connections from IP addresses that create too many address requests (and in any case should throttle it per IP address - not MVP problem to solve).

Once the address is confirmed the chat client will notify the user, and the user can configure in the client which + extensions should be allowed (in case catch-all was registered with the server) - not MVP. The user can now share this address with other users.

Sending contact requests to the address

- the sender's client should determine which identity server to use - either via DNS lookup and in case the server is absent, via our identity server.
- the sender's client should send message to the identity server's unsecured public queue. From SMP syntax perspective it should probably be SMP confirmation, TBC, the message itself will be `i.addr.req <email>` - note that this does not expose sender's identity in any way.
- the identity server should respond with a message queue provisioned specifically for this address with `i.addr.con <connection_request>`, where `connection_request` is the one created specifically for this address and contains recipient's public key.

The process above can be used by the address owner to periodically verify that the server does not substitute recipient's public key.

The server should always respond with `i.addr.con <connection_request>` even if the address is not registered, to avoid address-scanning - the response time should be consistent, so the server would have a pool of queues for such invalid responses.

- having received this response, the sender would send the message encrypted with the recipient's public key (or derived key from recipient's public key and sender's public key sent together with the message) e.g. `i.addr.hello <sender_address> <public_key>`
- the server forwards it to the recipient (the server cannot decrypt this message), with some anti-DDoS filters.
- recipient decrypts the message and repeats the same process with the sender's address. This message would be double encrypted, to avoid sharing recipient's address, in case sender used somebody else's address.

- if the sender used the address they control they would receive the confirmation from the recipient, and would be able to send this confirmation back at which point the recipient user will be notified.