SSB Rooms 2.0

Revision: 2021-02-10

Author: Andre Medeiros contact@staltz.com

Abstract

A room server is an SSB peer with privileged internet presence (for instance, not behind a NAT layer) which allows its clients to perform tunneled connections wich each other. For practical purposes, room clients seem to be connected to each other directly, but the room is an intermediary. Connections between server and client are end-to-end encrypted via secret-handshake, as well as in tunneled connections between room clients, so that the room server cannot eavesdrop on the payloads in the tunneled connections. This document describes new capabilities for rooms, such as user aliases, privacy modes, and tunneled authentication.

Table of contents

Stakeholders

Persons or organizations that are involved or engaged in or around room servers. They may hold responsibilities or powers, and may cause harm to other stakeholders when their responsibilities or powers are abused. They hold interest in engaging with other stakeholders while managing the risk for harm associated with engagement. Harm mitigation such as Privacy modes is important when discussing stakeholders.

Room admin

Person or organization responsible for operating the room server, and has full access rights over the server's resources such as logs, disk, memory, etc. In other words, this person or organization physically owns the room server or has SSH access to the remote server hosted in some PaaS cloud provider.

Typically, the admin possesses an SSB ID (it's very common, but not necessarily always the case), and is also a moderator in the room.

Internal user

SSB user who accesses the room server and is considered *internal* because they have already joined the room and may even have registered an alias in the room.

Specification

Definition: an *internal user* of a room is any SSB ID for which the room grants a tunnel address. In other words, if an SSB ID is *reachable* via a tunneled connection through a room server, then they are considered an internal user of that room.

Becoming an internal user: read more about that in Joining a room.

External user

Any SSB user who is not an internal user of the room (i.e. do not have a usable tunnel address referencing the room), but may still interact with the room server in meaningful ways, for instance with tunneled connections, alias endpoints or alias consumption.

Moderator

A moderator is an internal user that has acquired special privileges in the web dashboard and actions allowed by the dashboard.

Moderators can use Moderator sign-in to access the dashboard.

Setup

There are different ways a room server can be configured.

Components

A room server is defined by several components, which are systems that enable features, some of these are optional and some are required.

Required

- Tunneled connection
- Tunnel addresses
- Privacy modes (at least the *Open* mode)
- Joining (for at least the *Open* mode)

Optional

- Other Privacy modes and respective ways of joining
- Internal user authentication
- Tunneled authentication
- Invite endpoint
- Web Dashboard and Sign-in with SSB
- Aliases (requires "Web Dashboard" and "Sign-in with SSB")

Privacy modes

A room server should allow the room admin or a moderator to configure which users can become internal user.

Specification

There are three strategies recommended as policies to join the room, known as privacy modes:

- **Open**: invite codes are openly known, similar to ssb-room v1
- **Community**: only internal users can invite external users to become an internal users
- **Restricted**: only moderators can invite external users to become an internal users, and aliases are not supported

Joining: To become a member of the room, peers need to join the room.

Config database

The configuration database holds basic administrative data, readable only by admins and (indirectly via the dashboard) by moderators.

Specification

The database should contain these data points:

- Which privacy mode is selected
- List of SSB IDs of moderators
- List of blocked SSB IDs
- Name of the room (a short string)
- Description for the room (not too long string)

Web Dashboard

This is a WWW interface that allows moderators to sign-in and perform some privileged actions. Internal users can also sign-in and perform basic actions such as create invites for other users to join.

Specification

The dashboard grants moderators with features and powers such as:

- Block SSB IDs from connecting with this room, meaning two things:
  - If they were an internal user, they get demoted to external user
  - Even if they were an external user, the room server will reject new attempts of secret-handshake connections
- Unblock SSB IDs that are blocked
- Nominate other internal users to become moderators too
- View the list of aliases according to the Alias database
- Revoke aliases by removing an entry from the Alias database
- Change the privacy mode of the room
- View other technical measurements such as bandwidth used, storage used by the databases, etc

The dashboard grants internal users basic features such as:

- Register an alias for themselves
- Revoke an alias for themselves
- Create an invite for external users to join the room if the room is running in Community mode

Security considerations

Malicious moderator

Moderators obviously hold some power, and this power may be abused through unfair blocks, unfair revoking of aliases. In many cases, fairness is subjective, and is understood to be an essential compromise of having moderation to begin with. So in this section we will focus our attention on unusual security issues with moderation.

A moderator has the right to nominate other internal users to become moderators, and this could lead to a proliferation of moderators, which increases the possibility that one of these moderators abuses their powers. On the other hand, there has been many maintainers and npm owners in the SSBC (e.g. 32 GitHub org members and 17 npm owners for the cornerstone `ssb-db` package), we also know that the presence of many moderators may also help to *decrease* the possibility of abuse, because asymmetry of privilege is reduced.

Sign-in with SSB

To access the WWW dashboard interface, internal users (including moderators) can use "sign-in with SSB ID".

Specification

An internal user known by its SSB ID `cid` is connected to the room via secret-handshake and muxrpc. A browser client is supposedly the same person or agent as the internal user and wishes to gain access to the web dashboard. All HTTP requests SHOULD be done with HTTPS.

The three sides (browser client, SSB peer, and room server) perform the following challenge-response authentication protocol, specified as a UML sequence diagram. We use the shorthands `cc`, `sr`, `sc`, and `cr` to mean:

- `cc`: "client's challenge"
- `sr`: "server's response"
- `sc`: "server's challenge"
- `cr`: "client's response"

The challenges, `cc` and `sc`, are 256-bit cryptographic nonces encoded in base64. The responses, `sr` and `cr`, are cryptographic signatures using the cryptographic keypairs that identify the server and the client, respectively, described below:

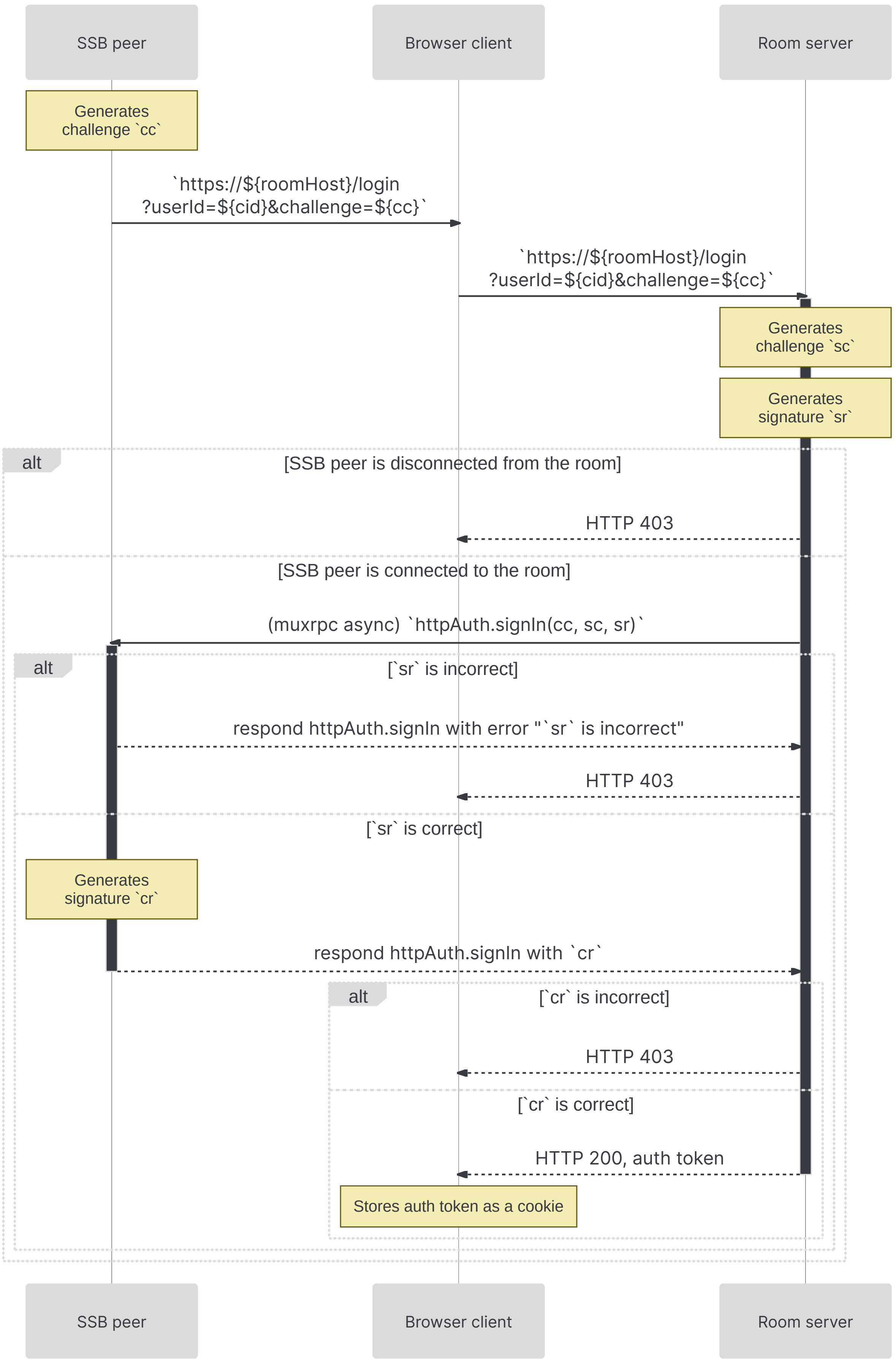
- `cid` is the client's identity from their cryptographic keypair
- `sid` is the servers's identity from their cryptographic keypair
- `cc` is a 256-bit nonce created by the client, encoded in base64
- `sc` is a 256-bit nonce created by the client, encoded in base64
- `sr` is the server's cryptographic signature of the string `=http-auth-sign-in:${cid}:${sid}:${cc}:${sc}` where `${x}` means string interpolation of the value `x`
- `cr` is the client's cryptographic signature of the string `=http-auth-sign-in:${cid}:${sid}:${cc}:${sc}` where `${x}` means string interpolation of the value `x`

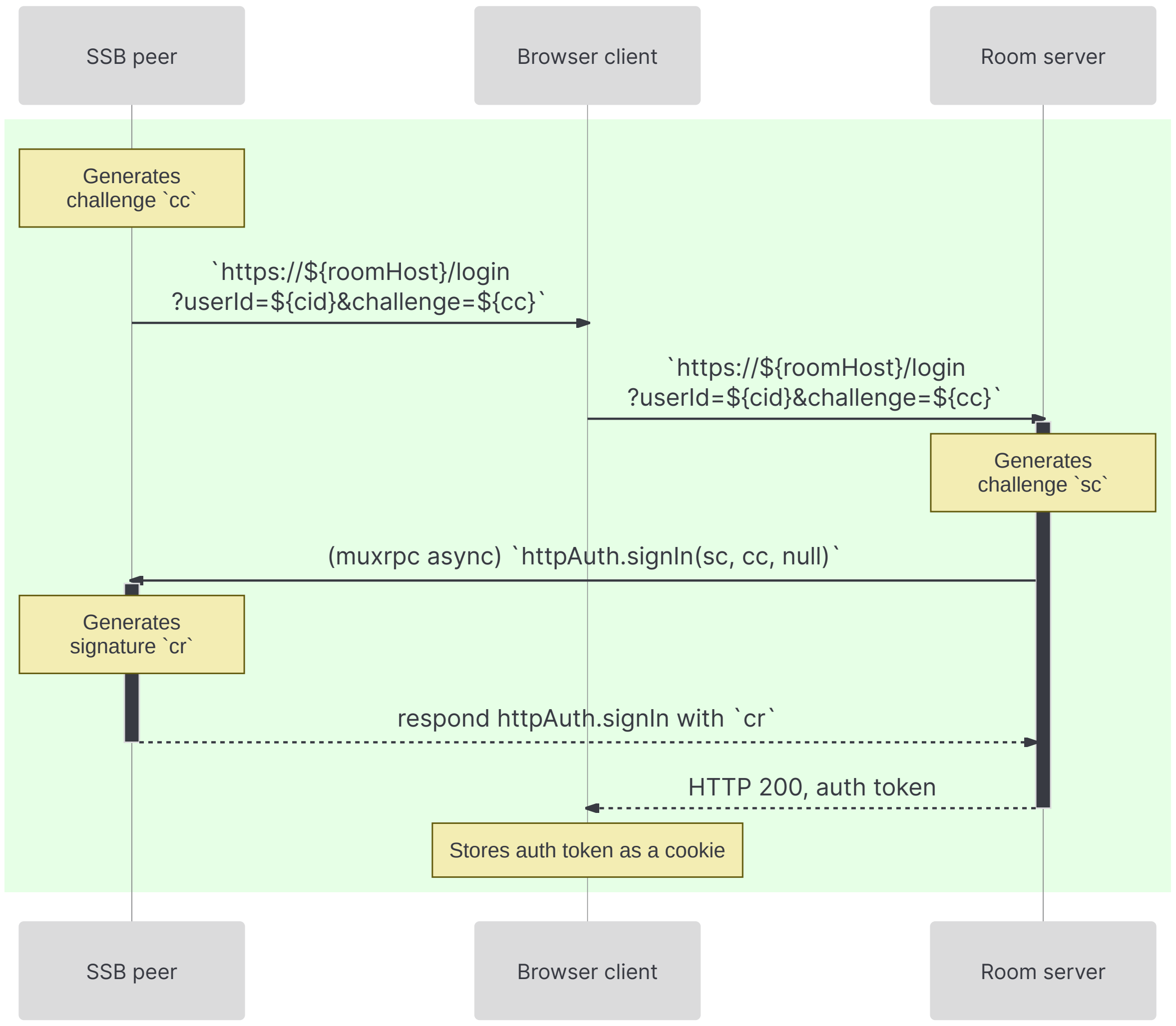
Both sides generate the nonces, but there are use cases where one side should start first. In other words, the challenge-response protocol described here can be either **client-initiated** or **server-initiated**.

Client-initiated protocol

In the client-initiated variant of the challenge-response protocol, the first step is the client creating `cc` and opening a web page in the browser. Then, the server attending to that HTTP request will call `httpAuth.signIn(cc, sc, sr)` on the client SSB peer.

The UML sequence diagram for the whole client-initial protocol is shown below:


SSB peer
Browser client
Room server
Generates challenge `cc`
`https://${roomHost}/login ?userId=${cid}&challenge=${cc}`
`https://${roomHost}/login ?userId=${cid}&challenge=${cc}`
Generates challenge `sc`
Generates signature `sr`
alt
[SSB peer is disconnected from the room]
HTTP 403
[SSB peer is connected to the room]
(muxrpc async) `httpAuth.signIn(cc, sc, sr)`
alt
[`sr` is incorrect]
respond httpAuth.signIn with error "`sr` is incorrect"
HTTP 403
[`sr` is correct]
Generates signature `cr`
respond httpAuth.signIn with `cr`
alt
[`cr` is incorrect]
HTTP 403
[`cr` is correct]
HTTP 200, auth token
Stores auth token as a cookie
SSB peer
Browser client
Room server

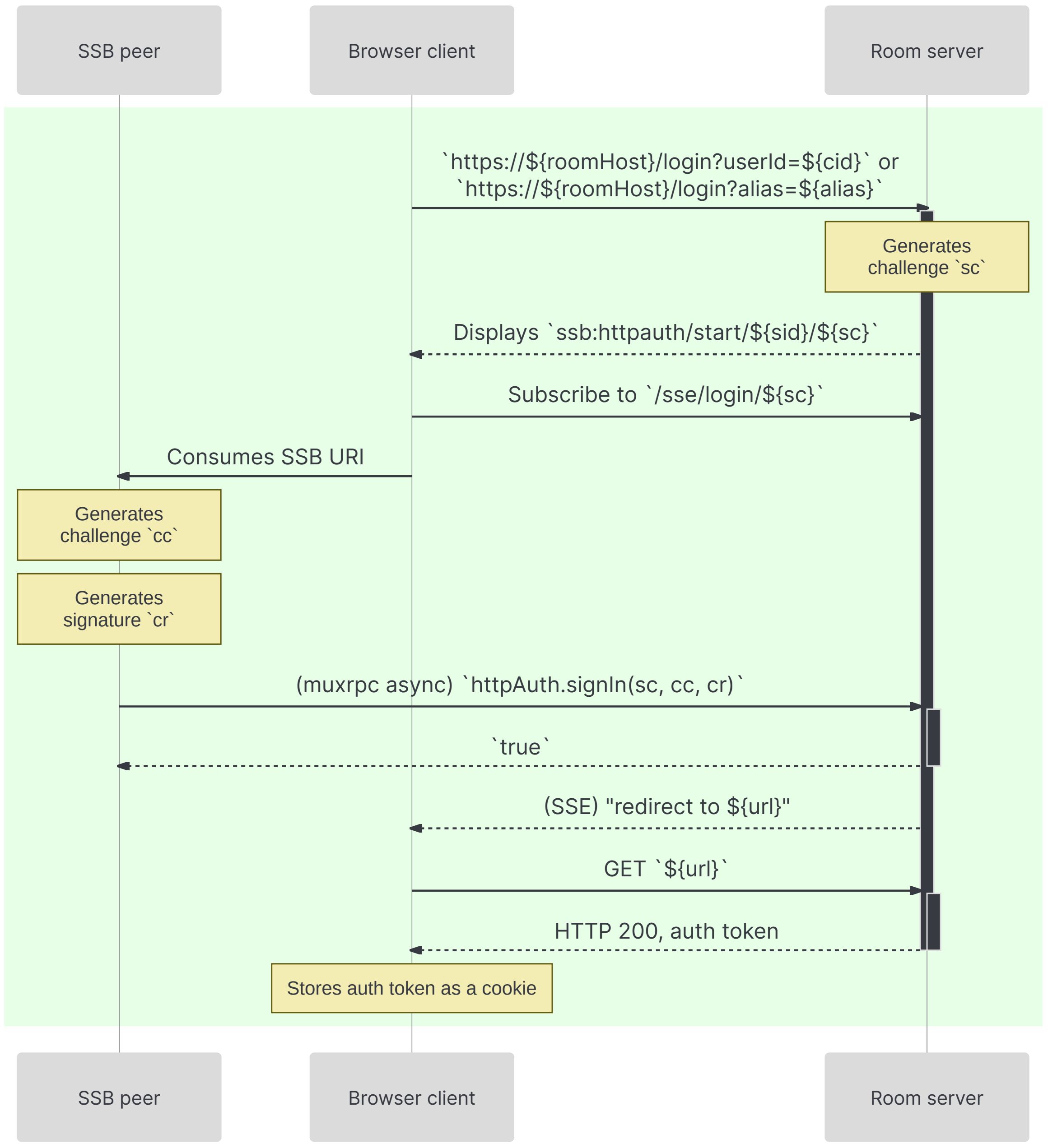

SSB peer
Browser client
Room server
Generates challenge `cc`
`https://${roomHost}/login ?userId=${cid}&challenge=${cc}`
`https://${roomHost}/login ?userId=${cid}&challenge=${cc}`
Generates challenge `sc`
(muxrpc async) `httpAuth.signIn(sc, cc, null)`
Generates signature `cr`
respond httpAuth.signIn with `cr`
HTTP 200, auth token
Stores auth token as a cookie
SSB peer
Browser client
Room server

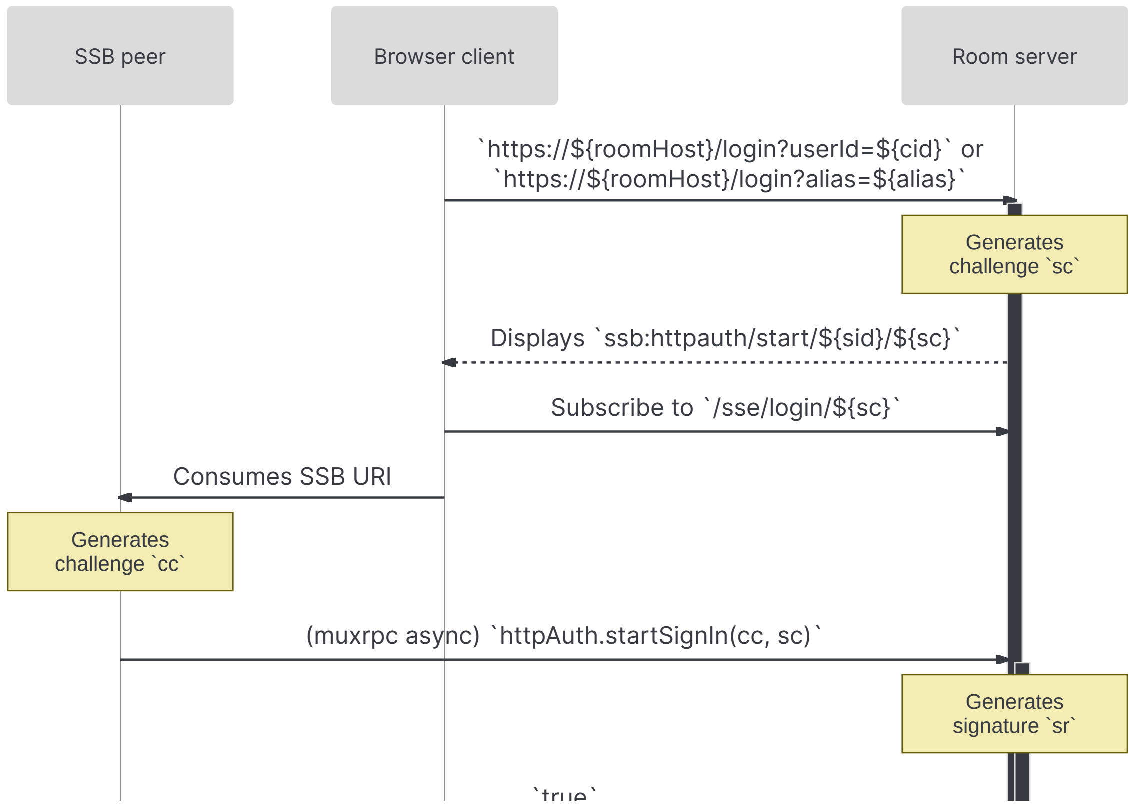
Server-initiated protocol

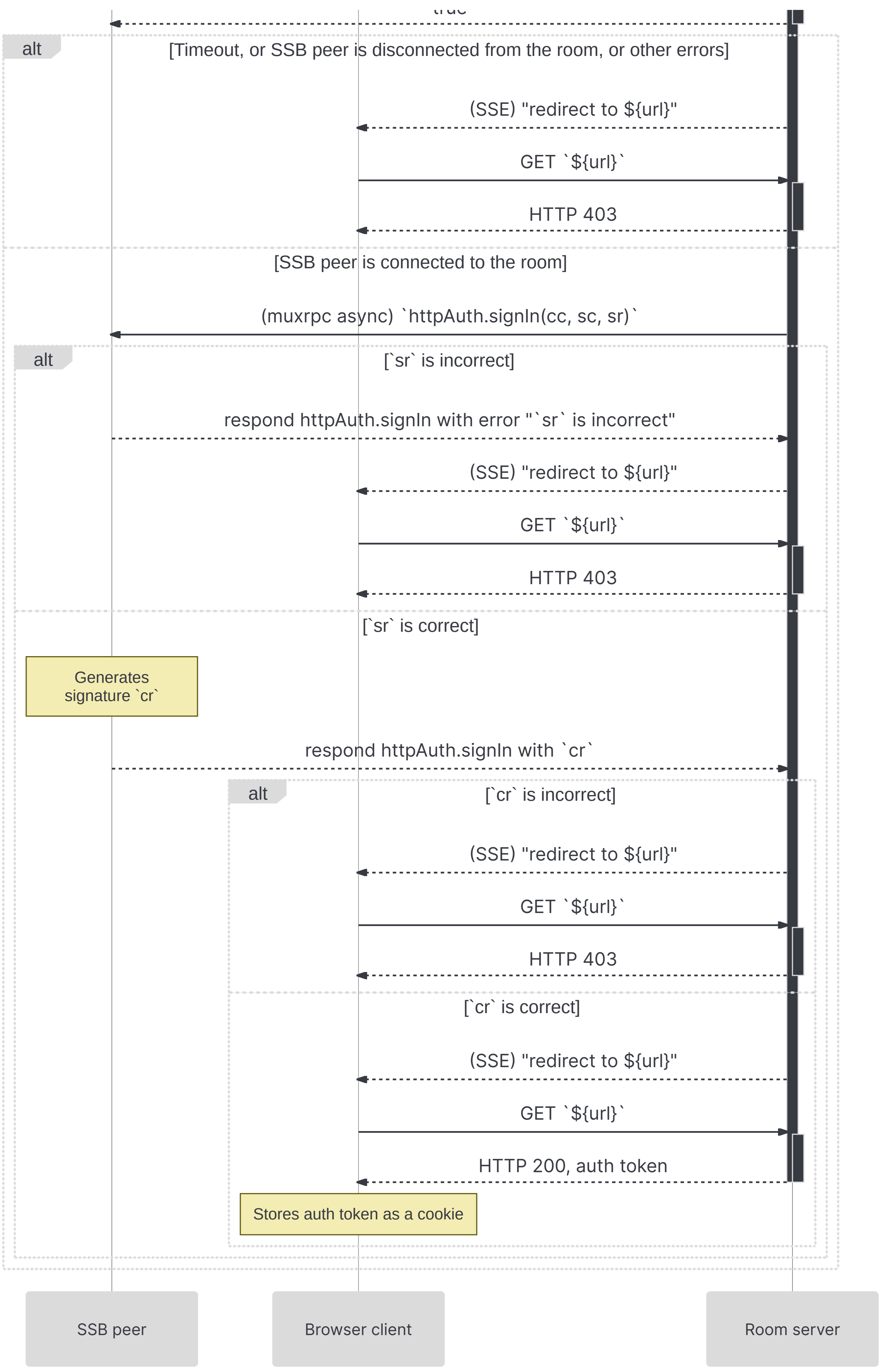
In the server-initiated variant of the challenge-response protocol, the first step is the browser requesting the server to login with a certain `cid` (or `alias`, which the server knows how to map to a `cid`). The server answers the browser, which in turn displays an SSB URI which the SSB peer knows how to open.

The primary difference between this variant and the previous one is the muxrpc async RPC `httpAuth.requestSignIn` which is used for the SSB peer to inform the room peer about the `cc`. Afterwards, the protocol is similar to the server-initiated one, with the minor addition of Server-Sent Events between the browser and the room.

The UML sequence diagram for the whole server-initial protocol is shown below:

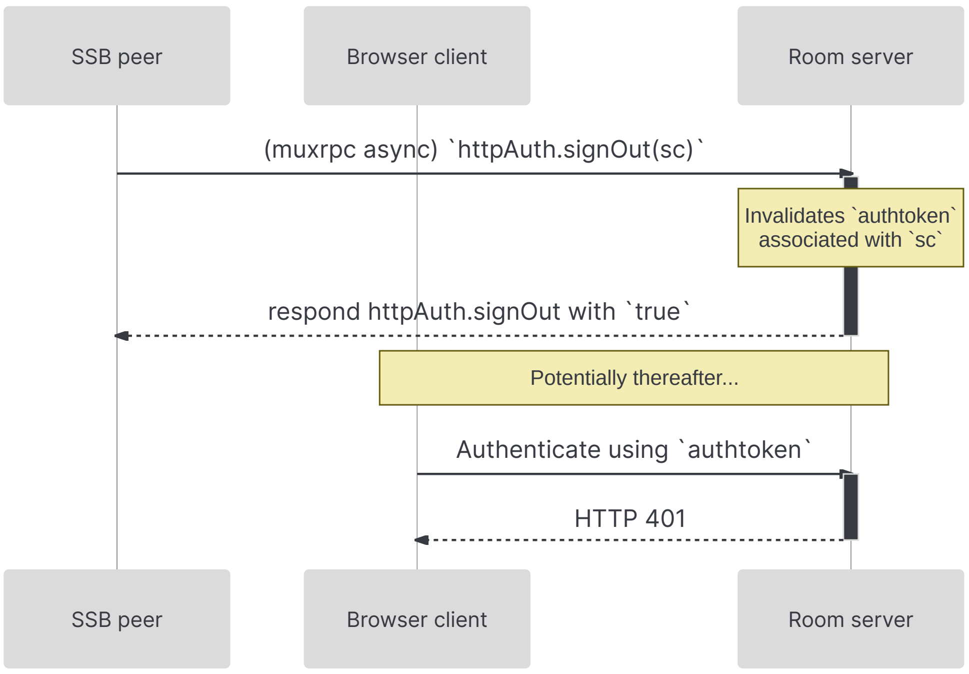

SSB peer
Browser client
Room server
`https://${roomHost}/login?userId=${cid}` or `https://${roomHost}/login?alias=${alias}`
Generates challenge `sc`
Displays `ssb:httpauth/start/${sid}/${sc}`
Subscribe to `/sse/login/${sc}`
Consumes SSB URI
Generates challenge `cc`
Generates signature `cr`
(muxrpc async) `httpAuth.signIn(sc, cc, cr)`
`true`
(SSE) "redirect to ${url}"
GET `${url}`
HTTP 200, auth token
Stores auth token as a cookie
SSB peer
Browser client
Room server


SSB peer
Browser client
Room server
`https://${roomHost}/login?userId=${cid}` or `https://${roomHost}/login?alias=${alias}`
Generates challenge `sc`
Displays `ssb:httpauth/start/${sid}/${sc}`
Subscribe to `/sse/login/${sc}`
Consumes SSB URI
Generates challenge `cc`
(muxrpc async) `httpAuth.startSignIn(cc, sc)`
Generates signature `sr`

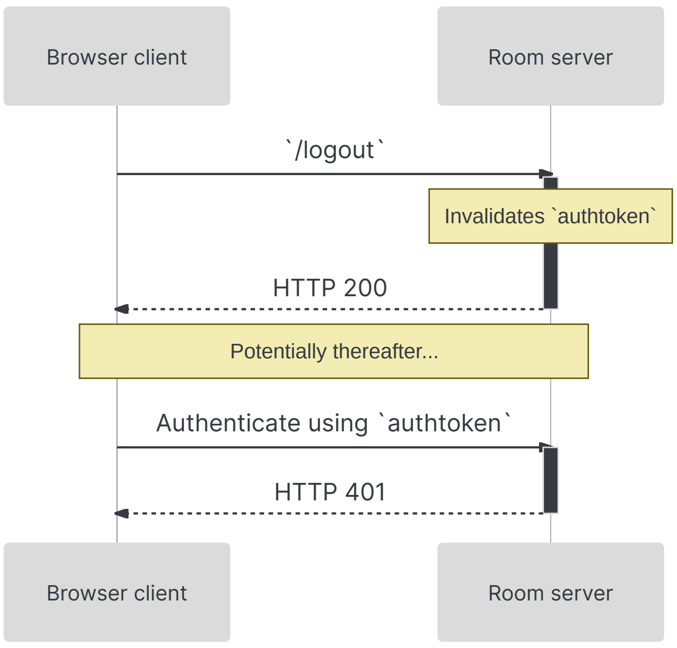

true
alt
[Timeout, or SSB peer is disconnected from the room, or other errors]
(SSE) "redirect to ${url}"
GET `${url}`
HTTP 403
[SSB peer is connected to the room]
(muxrpc async) `httpAuth.signIn(cc, sc, sr)`
alt
[`sr` is incorrect]
respond httpAuth.signIn with error "`sr` is incorrect"
(SSE) "redirect to ${url}"
GET `${url}`
HTTP 403
[`sr` is correct]
Generates signature `cr`
respond httpAuth.signIn with `cr`
alt
[`cr` is incorrect]
(SSE) "redirect to ${url}"
GET `${url}`
HTTP 403
[`cr` is correct]
(SSE) "redirect to ${url}"
GET `${url}`
HTTP 200, auth token
Stores auth token as a cookie
SSB peer
Browser client
Room server

Sign-out

An optional (but recommended) muxrpc API `httpAuth.signOut` on the Room server to allow the SSB peer to invalidate the auth token. See UML sequence diagram:


SSB peer
Browser client
Room server
(muxrpc async) `httpAuth.signOut(sc)`
Invalidates `authtoken` associated with `sc`
respond httpAuth.signOut with `true`
Potentially thereafter...
Authenticate using `authtoken`
HTTP 401
SSB peer
Browser client
Room server

The browser client also has the option of signing out with HTTP endpoints. This does not require a muxrpc call with the SSB peer. See UML sequence diagram:



Participation

Before peers can connect to each other via a room server, they first need to become members, i.e. internal users. This section describes the different protocols used for establishing internal user participation.

Joining

"Joining a room" means the process where an external user becomes an internal user.

Specification

The joining process is different for each Privacy mode:

- **Open:**
  1. An external user, Alice, acquires the open *invite code* either through the room's public website or via other means
  2. Alice consumes the invite code in her SSB app that supports being a room client
  3. The room accepts the connection from Alice and immediately grants her a tunnel address

    4. Alice has become an internal user
- **Community:**
    1. An internal user, Bob, signs into the room's web dashboard where he creates a one-time invite code in the form of an invite endpoint, provided on the dashboard provides.
    2. Bob informs an external user, Alice, of the invite code
    3. Alice consumes the invite code in their SSB app that supports being a room client
    4. The room checks whether the invite code is valid and has not yet been consumed
        1. If it is invalid or has been consumed, reply to Alice with an error
        2. Else, proceed (below)
    5. The room accepts the connection from Alice and immediately grants her a tunnel address
    6. Alice has become an internal user
    7. The room stores Alice's SSB ID in the Internal user registry
- **Restricted:**
    1. A moderator, Carla, signs into the room's web dashboard where she creates a one-time invite code in the form of an invite endpoint, provided on the dashboard.
    2. Bob informs an external user, Alice, of the invite code
    3. Alice consumes the invite code in their SSB app that supports being a room client
    4. The room checks whether the invite code is valid and has not yet been consumed
        1. If it is invalid or has been consumed, reply to Alice with an error
        2. Else, proceed (below)
    5. The room accepts the connection from Alice and immediately grants her a tunnel address
    6. Alice has become an internal user
    7. The room stores Alice's SSB ID in the Internal user registry

Internal user registry

The *internal user registry* is a database the room manages, keeping records of which SSB users are internal users. It is a simple list or table, where each entry refers to an internal user, and must contain at least the SSB ID for that user.

Internal user authentication

In rooms where the privacy mode is not *open*, not all SSB users who connect to the room are internal users. The room thus needs a way to authenticate the user before granting them a tunnel address.

Specification

When the room receives a secret-handshake incoming connection from Alice, it checks the internal user registry, looking for entry in the registry corresponding to Alice's ID. If there is an entry, the room allows the incoming connection to stay alive, and grants Alice a tunnel address. Otherwise, the room allows the connection but does not grant Alice a tunnel address.

We need the connection to remain up even in the event of internal user authentication failing, because there are other muxrpc APIs that the room should allow external users to call, such as when consuming an invite (to become an internal user) or to perform alias consumption.

Security considerations

Malicious external user

In the case of a room configured with privacy modes *Restricted*, the internal users of this room may want to be shielded from any external user gathering data about them, such as resolving aliases via web endpoints. The room needs to allow the external user to call muxrpc APIs, because the external user may be trying to join by consuming an invite. But in the case of a malicious external user, they may try to call other muxrpc APIs and so far this spec does not address how to protect against this possibility.

Malicious room admin

The room software could be modified by the room admin to not authenticate some users as internal users.

Invite endpoint

When joining a *Community* room or *Restricted* room, internal users create invites. The invite code is originally just random bytes encoded as hex, but can be transformed into an *invite link*, i.e. an HTTPS endpoint URL on the room server.

Example

1. Invite link is `https://scuttlebutt.eu/join?invite=39c0ac1850ec9af14f1bb73`
2. User opens that link in a browser, which redirects to `ssb:address/netshs/scuttlebutt.eu/8008/51w4nYL0k7mRzDGw20KQqCjt35y8qLiBNtWk3MX7ppo%3D?inviteType=room&inviteCode=39c0ac1850ec9af14f1bb73`
3. User's operating system opens that SSB URI in an SSB app, which then communicates with the room server via muxrpc, read more in Joining

Specification

1. Suppose the room is hosted at domain `roomHost` and it has generated an invite `inviteCode`
2. The invite link is `https://${roomHost}/join?invite=${inviteCode}`
3. Once a web visitor opens that link, then they are presented with the SSB URI `ssb:address/${roomMsAddr}?inviteType=room&inviteCode=${inviteCode}` where `${roomMsAddr}` consititutes the room's multiserver address in an SSB URI friendly format
    1. If the operating system detects an installed SSB app that recognizes the `ssb` scheme, the SSB app handles this URI
    2. Else, the website at displays instructions how to install an SSB app
        - For instance, there could be an app store URL (see this technical possibility) to install Manyverse which further redirects to this SSB URI, maybe may have to rely on a fixed URL (for Manyverse to register in its manifest) such as `join.manyver.se`. Same idea for other mobile apps, say "Imaginary App" using the fixed URL "join.imaginary.app". Desktop apps are different as they can be installed without an app store.
4. The SSB app knows how to parse the SSB URI into inputs necessary for joining the room

Security considerations

Malicious web visitor

A web visitor, either human or bot, could attempt brute force visiting all possible invite endpoints, in order to force themselves to become an internal user. However, this could easily be mitigated by rate limiting requests by the same IP address.

Tunnel addresses

To establish a tunneled connection, the peer initiating it must know the *tunnel address* of the peer at the other side of the tunnel.

Specification

A tunnel address is a string conforming to the multiserver-address grammar. We say that "room M *grants* peer A a tunnel address" when room M allows other peers to request and establish tunneled connections with peer A, using the tunnel address to identify peer A.

It consists of three parts and `:` as separators in between:

- `tunnel` as a constant tag
- SSB ID of the intermediary peer
- SSB ID of the target peer

Example

Without spaces nor newlines:

```
tunnel:@7MG1hyfz8SsxlIgansud4LKM57IHIw2Okw
/hvOdeJWw=.ed25519:@1b9KP8znF7A4i8wnSevBSK
2ZabI/Re4bYF/Vh3hXasQ=.ed25519
```

The tunnel address, being a multiserver address, can also contain a *transform* section, such as the common `shs` transform (without spaces nor newlines):

```
tunnel:@7MG1hyfz8SsxlIgansud4LKM57IHIw2Okw
/hvOdeJWw=.ed25519:@1b9KP8znF7A4i8wnSevBSK
2ZabI/Re4bYF/Vh3hXasQ=.ed25519~shs:1b9KP8z
nF7A4i8wnSevBSK2ZabI/Re4bYF/Vh3hXasQ=
```

Tunneled connection

A tunneled connection is an indirect connection between two peers assisted by an intermediary peer. Ideally, two peers could always connect with each other directly, but they often have unstable IP addresses behind NATs and firewalls, making it difficult to consistently and reliably establish connections. The purpose of the intermediary peer is to improve connection reliability, because these intermediary peers can be privileged nodes with public IP addresses, such as from hosting services.

Specification

Tunneled connections in SSB originated from the proof-of-concept ssb-tunnel module. Suppose A and B are clients of a intermediary server M. Peer A creates a conventional handshake connection to M, and waits to receive tunnel connections. Peer B creates a conventional secret handshake connection to M, and then requests a tunneled connection to A through that conventional connection (B-M). Then, M calls A, creating a tunneled connection where one end is attached to A and the other end is attached to B's request. Finally, B uses the secret handshake to authenticate A.

Notice that for the intermediary M, peer A is the server and B is the client (client calls, server answers) but M is just the portal. The tunneled connection is inside the outer (conventional) connections, which means it is encrypted twice with box stream. This means A and B can mutually authenticate each other, and M cannot see the content of their connection.

Diagram:

```
,---,      ,---,     ,---,
|   |----->|   |<----|   |
| A |<=====| M |<====| B |
|   |----->|   |<----|   |
`---`      `---`     `---`
```

The arrows represent the direction of the connection – from the client, pointing to the server. Notice the M<=B connection is the same direction as the M←B outer connection, but the A<=M connection is the opposite direction as the A→M outer connection.

Security considerations

Malicious room admin

The room admin could log and track all connection sessions for every tunneled connection, thus tracking the **IP addresses**, **timestamps**, **durations**, and **bandwidth** of interactions between internal users. The room admin could track which SSB users are interested in connecting with internal users, i.e. they can gather **social interest metadata**, which could be used to create a draft of a portion of the social graph.

That said, because of encrypted tunneled secret-handshake channels, the room admin could not know the contents of data transmitted between the internal users.

Tunneled authentication

Tunneled authentication is about making sure that SSB peers on the opposite end of a tunneled connection only allow the connection to occur if they follow the peer on the other side. Thus we need a way for peers to know who wants to open a tunneled connection and we should facilitate mutual follows to occur so that peers only create tunneled connections imbued with mutual interest.

Specification

Tunneled friend authentication is an algorithm or protocol that applies automatically without any user input from either end of the secret-handshake channel. This protocol should not apply for the intermediary peer, that is, the room server.

When Alice receives a tunneled secret-handshake incoming connection from Bob, she automatically allows it if Alice checks that she follows Bob, or automatically cancels the connection if Alice checks that she does not follow Bob (or blocks Bob). Same is true reciprocically: Bob applies this rule for incoming connections from Alice.

Thus tunneled authentication **requires mutual follows** ("friendship") before establishing a functioning tunneled connection.

When a denial of connection occurs, the peer that received the connection should be able to see (and thus locally log): (1) SSB ID of the intermediary peer (room) used, (2) SSB ID of the origin peer behind the intermediary, (3) (optionally) the address (tunnel address or alias endpoint URL) of the origin peer.

The user that received the denied connection can then see this fact in their SSB app, and then they can make a conscious choice to either (1) follow the origin peer, or (2) connect to the origin peer (if (3) from the previous paragraph existed), or both.

Implementation notes

Note that in current room server implementation in JavaScript, `opts.origin` in the room is calculated from secret-handshake, so it can be trusted to be authentic.

For the next version of rooms, if we want `opts.origin` to also contain the origin peer's address (ssb-tunnel address or alias endpoint), then we need other means of verifying that the origin address is authentic. E.g. if it's an alias endpoint URL, maybe the receiving peer visits the alias JSON endpoint then consumes the alias, or maybe the receiving peer takes the ssb-tunnel address and verifies that the ID matches with the secret-handshake-given ID.

Alias

An alias (also known as "room alias") is a string that identifies an internal user, designed to be short and human-friendly, similar to email addresses and Mastodon WebFinger addresses. The purpose of aliases is give improve the user experience of accurately (1) **identifying** the internal user, (2) **locating** the internal user at a room server for the purpose of establishing a connection with them.

As an example, suppose Alice is an internal user of the room "Scuttlebutt EU". The room's domain is `scuttlebutt.eu` and Alice's alias is `alice`. Alice's alias endpoint is thus `alice.scuttlebutt.eu`.

In short,

- Anyone can access an alias web endpoint
- Internal users can register and revoke aliases
- Internal users and external users who visit a target user's alias endpoint can consume it in order to connect with the target internal user
- The room admin has read/write access to the alias database
- Moderators can remove alias entries from the alias database

Alias string

An internal user's alias, also known as "alias string", is used to uniquely (unique within the room server only) identify that internal user. This string is useful only within the context of the room, i.e. not globally identifiable.

Example

Suppose Alice is an internal user of the room "Scuttlebutt EU". Alice's alias could be one of these strings (non-exhaustive list):

- `alice`
- `alice1994`

- `alice_94`

Specification

The string should satisfy the same rules as domain "labels" as defined in RFC 1035.

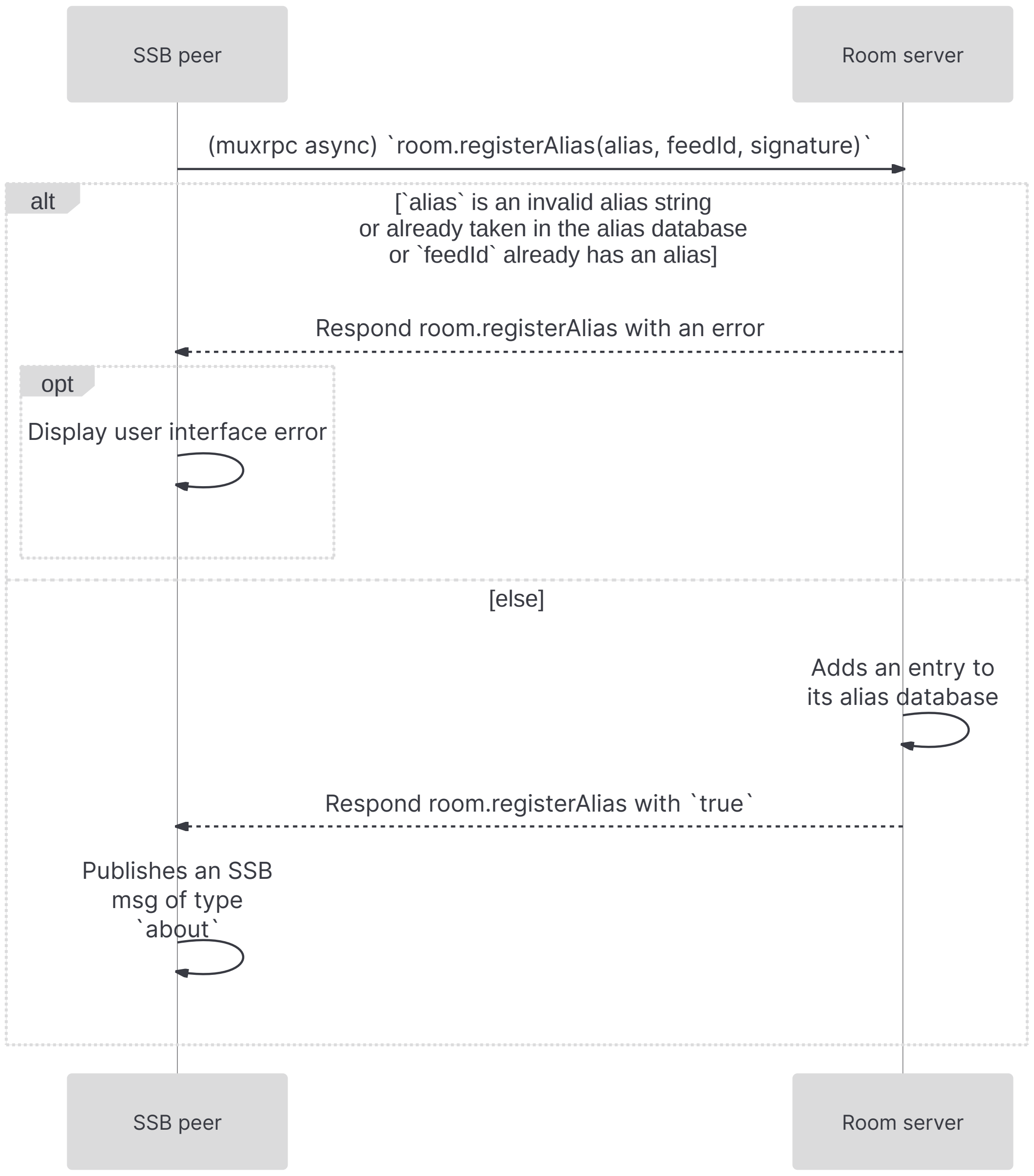
Alias registration

An internal user who does not have an alias in the current room server can choose to register an alias. Not all internal users need to have aliases, so the process described here is optional.

Specification

1. An internal user with SSB ID `feedId` and a room server with SSB ID `roomId` are connected to each other via secret-handshake
2. The internal user chooses a `alias` as a candidate alias string
3. The internal user calls a specific muxrpc `async` API `room.registerAlias(alias, feedId, signature, callback)` where `signature` is a cryptographic signature of the string `=room-alias-registration:${roomId}:${feedId}:${alias}` using `feedId`'s cryptographic keypair, read more about it in the alias database spec
4. The room, upon receiving the `room.registerAlias` muxrpc call, checks whether that `alias` is valid (see spec in Alias string)
    1. If it is invalid, respond `room.registerAlias` with an error
    2. Else, proceed (below)
5. The room checks whether there already exists an entry in the Alias database associated with this `feedId`
    1. If there is, respond `room.registerAlias` with an error
    2. Else, proceed (below)
6. The room checks whether there already exists an entry in the Alias database with the *key* `alias`
    1. If there is, respond `room.registerAlias` with an error
    2. Else, proceed (below)
7. The room adds an entry to its Alias database for `key=alias` & `value=feedId+sig`
8. The room responds `room.registerAlias` with `true`, indicating success
9. The internal user receives the room's response to `room.registerAlias`
    1. If it is an error, then (optionally) display a user interface failure to register the alias
    2. If it is `true`, then publish an SSB msg of type `about` with a field listing all its aliases for various rooms, where this specific `alias` is included. The specific schema of the message type is an application-level concern

The above algorithm is also provided below as a UML sequence diagram:


SSB peer
Room server
(muxrpc async) `room.registerAlias(alias, feedId, signature)`
alt
[`alias` is an invalid alias string
or already taken in the alias database
or `feedId` already has an alias]
Respond room.registerAlias with an error
opt
Display user interface error
[else]
Adds an entry to
its alias database
Respond room.registerAlias with `true`
Publishes an SSB
msg of type
`about`
SSB peer
Room server

Security considerations

Malicious internal user

The reason why there can be only one alias for SSB ID is to prevent a malicious internal user from exhausting many or all possible aliases in case the room accidentally allows such malicious user to become an internal user. Arguably, some room implementations could choose to relax this choice, perhaps to allow different aliases for an internal user, that covers typographic mistakes such as `aliec`, `alicce`. For the time being, it seems sensible that each internal user can receive only one alias.

Malicious room admin

The room admin could reply with errors when technically the muxrpc should have succeeded, e.g. pretending that the `alias` candidate is invalid or pretending that it's already registered.

Alias revocation

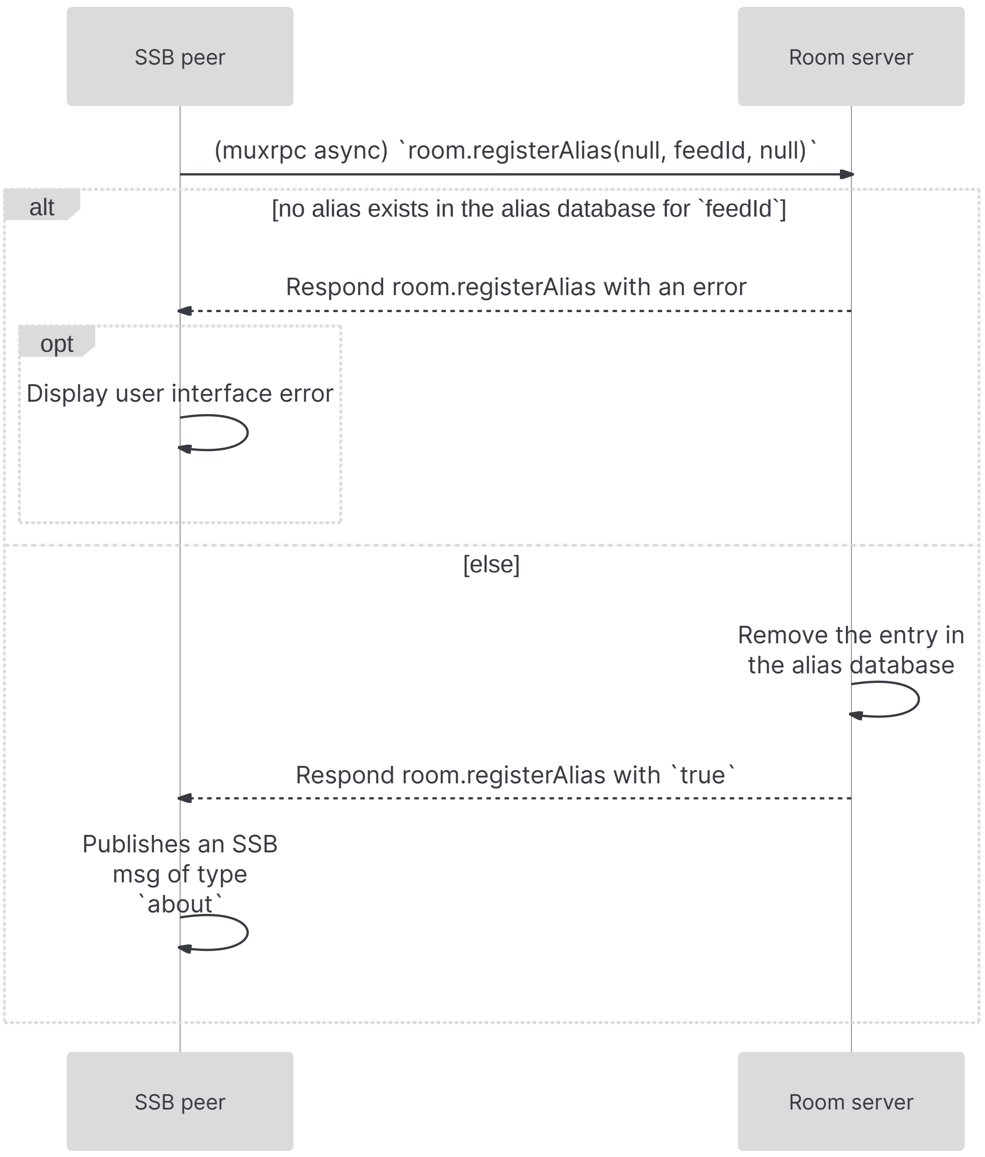
When an internal user who has registered no longer wishes to have that alias associated with them anymore, they can perform *alias revocation* to remove that alias from the alias database.

Specification

1. An internal user with SSB ID `feedId` and a room server with SSB ID `roomId` are connected to each other via secret-handshake
2. The internal user calls a specific muxrpc `async` API `room.registerAlias(null, feedId, null, callback)`

3. The room, upon receiving the `room.registerAlias` muxrpc call, checks whether there exists an entry in the Alias database associated with `feedId`
    1. If there is no entry, respond `room.registerAlias` with an error
    2. Else, proceed (below)
4. The room adds an entry to its Alias database for `key=alias` & `value=feedId+sig`
5. The room removes the entry from the Alias database associated with `feedId`
6. The room responds `room.registerAlias` with `true`, indicating success
7. The internal user receives the room's response to `room.registerAlias`
    1. If it is an error, then (optionally) display a user interface failure to revoke the alias
    2. If it is `true`, then publish an SSB msg of type `about` with a field listing all its aliases for various rooms, where this specific `alias` is no longer listed. The specific schema of the message type is an application-level concern

The above algorithm is also provided below as a UML sequence diagram:


SSB peer
Room server
(muxrpc async) `room.registerAlias(null, feedId, null)`
alt
[no alias exists in the alias database for `feedId`]
Respond room.registerAlias with an error
opt
Display user interface error
[else]
Remove the entry in the alias database
Respond room.registerAlias with `true`
Publishes an SSB msg of type `about`
SSB peer
Room server

Security considerations

Malicious room admin

The room admin could refuse to remove the database entry, or could delete the database entry at will (before the internal user performs revocation). In other words, the internal user does not ultimately have power over the deletion of the alias entry from the alias database, it must trust the room admin regarding deletion.

Alias consumption

When an SSB user (external or internal) is connected to the room, and knows of another internal user's alias, they can perform *alias consumption*. After consumption is completed successfully, they authentically obtain the target user's SSB ID and can use it to start a tunneled connection.

Specification

The input for the consumption algorithm is the response from the web endpoint, which is (either through JSON or SSB URI): the room's multiserver `address`, `roomId`, `userId`, `alias`, and `signature`.

1. The SSB user verifies that the `signature` authentically matches `roomId`, `userId` and `alias`
   1. If it is an invalid signature, interrupt alias consumption with a failure indicating that the alias association to the internal user `userId` was probably forged
   2. Else, proceed (below)
2. The SSB user acting as a client connects to the room's `address` and establishes a muxrpc connection
3. The client can now use `userId` to initiate a tunneled connection with them
4. (Optional and recommended) The client *follows* the `userId`, see tunneled authentication

Web endpoint

Once an alias is registered, it enables any web user to visit a web endpoint on the room server dedicated to that alias, for the purpose of telling the visitor what SSB ID does the alias resolve to, and with instructions on how to install an SSB app if the visitor doesn't have it yet.

The goal of this endpoint is to help any SSB user *locate and identify* the alias' owner by resolving the alias to: (1) the room's multiserver address, (2) the owner's SSB ID, and (3) a cryptographic signature that proves the owner associated themselves with that alias. This web endpoint is very valuable to onboard new SSB users being invited by an internal user.

Prior art: This endpoint should be in many ways similar to the Telegram `https://t.me/example` service for the username `@example`, also capable of redirecting the web visitor to a scheme `tg` URI `tg://resolve?domain=example`, which Telegram apps know how to parse and open the target user's profile screen.

Specification

This specification does not apply if the privacy mode is *Restricted*. This web endpoint is available only if the privacy mode is *Open* or *Community*.

If the alias `${alias}` is registered at the room `${roomHost}` for a certain `${userId}`, then the room's HTTP endpoint `https://${alias}.${roomHost}` SHOULD respond with HTML which:

- Informs users how to install an SSB app that can correctly consume room aliases
- Renders a "Connect with me" button linking to an SSB URI (see below)
- The page automatically redirects (when the browser supports it) to an SSB URI (see below)
- The alias SSB URI is `ssb:address/${roomMsAddr}?roomId=${roomId}&userId=${userId}&alias=${alias}&signature=${signature}`, in other words there are 2 components:
  - `ssb:address/${roomMsAddr}`, comprising the **path component** of the URI, is the room's multiserver address
  - The **query component** of the URI with the following 4 parts:
    - `roomId=${roomId}`, the room's SSB ID
    - `userId=${userId}`, the SSB ID of the alias's owner
    - `alias=${alias}`, the alias string
    - `signature=${signature}`, the alias's owner signature as described in the alias database

As an additional endpoint for programmatic purposes, the HTTP endpoint `https://${alias}.${roomHost}/json` MUST respond with a JSON body with the following schema:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://github.com/ssb-ngi-pointer/rooms2",
  "type": "object",
  "properties": {
    "address": {
      "title": "Multiserver address",
      "description": "Should conform to https://github.com/ssbc/multiserver-address",
      "type": "string"
    },
    "roomId": {
      "title": "Room ID",
      "description": "SSB ID for the room server",
      "type": "string"
    },
    "userId": {
      "title": "User ID",
      "description": "SSB ID for the user owning the alias",
      "type": "string"
    },
    "alias": {
      "title": "Alias",
      "description": "A domain 'label' as defined in RFC 1035",
      "type": "string"
    },
    "signature": {
      "title": "Signature",
      "description": "Cryptographic signature covering the roomId, the userId, and the alias",
      "type": "string"
    }
  },
  "required": [
    "address",
    "roomId",
    "userId",
    "alias",
    "signature"
  ]
}
```

Example

Suppose the alias is `alice`, registered for the user ID `@FlieaFef19uJ6jhHwv2CSkFrDLYKJd/SuIS71A5Y2as=.ed25519` at the room with host name `scuttlebutt.eu`. Then the alias endpoint `https://alice.scuttlebutt.eu` responds with the following SSB URI (without spaces nor newlines):

```
ssb:address/netshs/scuttlebutt.eu/8008/51w4nYL0k7mRzDG
w20KQqCjt35y8qLiBNtWk3MX7ppo%3D?roomId=ed25519%2F51w4nY
L0k7mRzDGw20KQqCjt35y8qLiBNtWk3MX7ppo%3D&userId=ed25519
%2FFlieaFef19uJ6jhHwv2CSkFrDLYKJd_SuIS71A5Y2as%3D&alias
=alice&signature=yNDgrVOLm6sMUHdvnbFUQYgLkCGiOKrpP9KiBv
lrzvmxTNt3d0MNTf%2BSLMIxgxf00S5fKAlG2%2FC5NTE0Zq1Mmg%3D
%3D
```

Note the multiserver address for the room is written in an SSB URI friendly format, `ssb:address/netshs/${host}/${port}/${pubkey}`, that can be converted to canonical multiserver format `net:${host}:${port}~shs:${pubkey}`.

The JSON endpoint `https://alice.scuttlebutt.eu/json` would respond with the following JSON:

```
{
  "address": "net:scuttlebutt.eu:8008~shs:51w4nYL0k7mRzDGw20KQqCjt35y8qLiBNtWk3MX7ppo=",
  "roomId": "@51w4nYL0k7mRzDGw20KQqCjt35y8qLiBNtWk3MX7ppo=.ed25519",
  "userId": "@FlieaFef19uJ6jhHwv2CSkFrDLYKJd/SuIS71A5Y2as=.ed25519",
  "alias": "alice",
  "signature": "yNDgrVOLm6sMUHdvnbFUQYgLkCGiOKrpP9KiBvlrzvmxTNt3d0MNTf+SLMIxgxf00S5fKAlG2/C5NTE0Zq1Mmg=="
}
```

Security considerations

Malicious web visitor

A web visitor, either human or bot, could attempt brute force visiting all possible alias endpoints, in order to build a dataset of all SSB IDs and claimed aliases gathered at this room, potentially tracking profiles of these SSB IDs. Malicious web visitors can also attempt to connect with these target IDs as victims, and may use social engineering or impersonation tactics during tunneled authentication.

Malicious room admin

The room admin could tamper with the alias database and provide fake information on this web endpoint, e.g. that a certain alias was claimed by a certain users. Although the database signature exists to prevent this type of tampering, it is only verified when performing alias consumption. For web visitors who only want to know which SSB ID corresponds to an alias, and only that, these visitors must trust the room administrator, who could provide inauthentic information.

Alias database

This is a database that stores all aliases that were registered by internal users.

Example

The following is a mock up of a key-value store:

Key	Value
`alice`	`@FlieaFef19uJ6jhHwv2CSkFrDLYKJd/SuIS71A5Y2as=.ed25519` plus signature
`bob`	`@25WfId3Vx/gyMAZqCyZzhtW4iPtUVXB/aOMYbq44P4c=.ed25519` plus signature
`carla`	`@dRE+jzKo0VWX6JbcSVATyOvFlbjCNwPWNzQLkTGenac=.ed25519` plus signature
`daniel`	`@SMMgb4bZAgRgtAPdMw4loQeZL9lQgsRDi+xin0ZDzAg=.ed25519` plus signature

Specification

This can be a simple persistent key-value store, such as Leveldb.

- Each **Key** is an alias string
- Each **Value** is a string that encodes two things:
  - SSB identity of the internal user
  - A cryptographic signature that covers **all these**
    - the room server's ID, i.e. `roomId`
    - the SSB ID, i.e. `userId`
    - alias string, i.e. `alias`

The signature is applied on the following string: `=room-alias-registration:${roomId}:${userId}:${alias}`, known as the *Alias confirmation*, see example (without spaces nor newlines):

```
=room-alias-registration:@51w4nYL0k7mRzDGw20KQqCjt35
y8qLiBNtWk3MX7ppo=.ed25519:@FlieaFef19uJ6jhHwv2
CSkFrDLYKJd/SuIS71A5Y2as=.ed25519:alice
```

where

- `roomId` is `@51w4nYL0k7mRzDGw20KQqCjt35y8qLiBNtWk3MX7ppo=.ed25519`
- `userId` is `@FlieaFef19uJ6jhHwv2CSkFrDLYKJd/SuIS71A5Y2as=.ed25519`
- `alias` is `alice`

The purpose of a cryptographic signature on the combined `roomId` & `userId` & `alias` is to make sure that the Room admin cannot tamper with the database to delegitimize its contents. This means that each key-value pair is certainly authored by the declared SSB ID, that is, neither the key (the alias) nor the value (the SSB ID) was modified by the Room admin.

Security considerations

Malicious room admin

The room admin can freely read or write to this database, they can create new entries, and so forth. If they modify an entry and thus break the validation of the signatures, other SSB users can detect this when verifying the signatures.

Thus the admin **cannot** effectively:

- Register a signed alias on behalf of an internal user
- Modify a registered alias made by internal users

But the admin **can**:

- Remove any registered key-value pairs from the database, essentially removing an alias
- Register signed aliases for fake users it has created itself

Malicious moderator

Similar considerations as with the room admin, but less powers. The malicious moderator *cannot* do the actions that the room admin cannot do (otherwise moderators would have more power than admins), but the one thing moderators can do is:

- Remove any registered key-value pairs from the database, essentially removing an alias

Appendix

List of new muxrpc APIs

- async
  - `room.registerAlias(alias, feedId, signature, callback)`
  - `httpAuth.signIn(sc, cc, cr, callback)`
  - `httpAuth.signOut(sc, callback)`

List of new SSB URIs

- `ssb:address/${multiserverAddress}`
  - Special case: `ssb:address/${roomMsAddr}?inviteType=room&inviteCode=${inviteCode}`
  - Special case: `ssb:address/${roomMsAddr}?roomId=${roomId}&userId=${userId}&alias=${alias}&signature=${signature}`
- `ssb:httpauth/start/${sid}/${sc}`