
unMessage Documentation

Release 0.1.0

Anemone Labs

January 24, 2017

1	What is it?	1
1.1	Overview	1
1.2	Installation	1
1.3	Graphical User Interface (GUI)	2
1.4	Command-line Interface (CLI)	8
1.5	unMessage Protocol	13
2	Other	17
2.1	Changelog	17
2.2	Feedback	17

What is it?

1.1 Overview

unMessage is a peer-to-peer instant messaging application designed to enhance privacy and anonymity.

Warning: unMessage is **alpha** software. While every effort has been made to make sure unMessage operates in a secure and bug-free fashion, the code has **not** been audited. Please do not use unMessage for any activity that your life depends upon.

1.1.1 Features

- Transport makes use of [Twisted](#), [Tor Onion Services](#) and [txtorcon](#)
- Encryption is performed using the [Double Ratchet Algorithm](#) implemented in [pyaxo](#) (using [PyNaCl](#))
- Authentication makes use of the [Socialist Millionaire Protocol](#) implemented in [Cryptully](#)
- Transport metadata is minimized by *Tor* and application metadata by the *unMessage Protocol*
- User interfaces are created with [Tkinter](#) (graphical) and [curses](#) (command-line)

1.2 Installation

Make sure that you have the following:

```
# If using Debian/Ubuntu
$ sudo apt-get install build-essential gcc libffi-dev python-dev tor tkinter

# If using Fedora
$ sudo yum install gcc libffi-devel python-devel redhat-rpm-config tor tkinter
```

If you use [pip](#) and [setuptools](#) (probably installed automatically with *pip*), you can easily install unMessage with:

```
$ sudo pip install unmessage
```

Launch unMessage with any of the commands:

```
$ unmessage-gui # graphical user interface (GUI)
$ unmessage-cli # command-line interface (CLI)
$ unmessage # last interface used
```

1.2.1 Updating

If you installed unMessage with *pip*, you can also use it for updates:

```
$ sudo pip install --upgrade unmessage
```

1.2.2 Usage

unMessage offers usage instructions for both interfaces: *Graphical User Interface (GUI)* and *Command-line Interface (CLI)*.

1.2.3 Persistence

All files used by unMessage are saved in `~/.config/unMessage/`

1.3 Graphical User Interface (GUI)

Launch unMessage's *GUI* with:

```
$ unmessage-gui
```

You are taken to the `Start Peer` tab and you are required to pick any name you wish to use and press `Start`:

Tor is launched and if this is the first time you use that name, your *Onion Service* and *Double Ratchet* keys are created and you are ready to receive and send requests to initialize conversations. unMessage displays this bootstrap process:

The `Copy` buttons at the top bar can be used to copy information the other peers need to send you requests. You must share both your **identity address** and **key**:

```
charlie@jt6zabesvrhxvhee.onion:50001 v4kU6s+NuJW/Znbjz0AxoI9Gvl1XDS5eiOTm6cE38E4=
```

1.3.1 Sending Requests

Press the `New chat` button at the top bar to open the `Request` window. Provide the **identity address** and **key** of the peer you wish to contact:

An **identity address** is provided in the format `<name>@<onion address>`, where the `<name>` is only a local identifier of the peer and you can pick any name you wish to call them.

1.3.2 Receiving Requests

Inbound requests are notified in a new window with the information of the peer who sent the request:

As mentioned previously, peer names are local and when accepting a request you can pick another one to call them instead of using the one they sent.

1.3.3 Chatting

unMessage creates tabs for each peer you have a conversation with. Within each tab, besides composing messages and sending (clicking `Send` or pressing the `Enter` key) there are some actions available.

New Chat Copy Identity Copy Key Copy Peer Copy Onion Quit

Start Peer

How will peers find you?

Name
charlie

Local Server Port (Optional)

Tor Port (Optional)

Tor Control Port (Optional)

Start

Fig. 1.1: Start Peer window

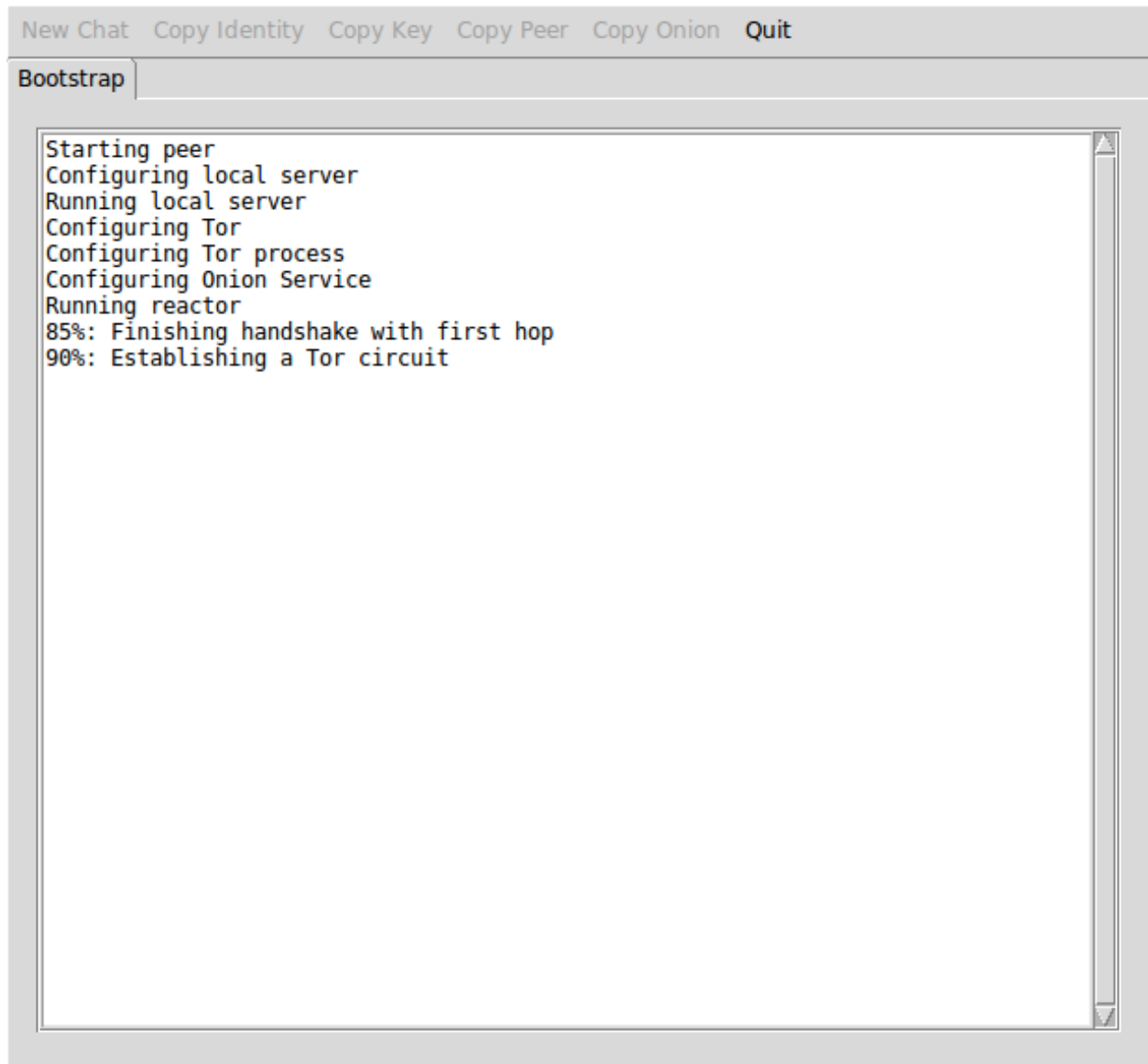
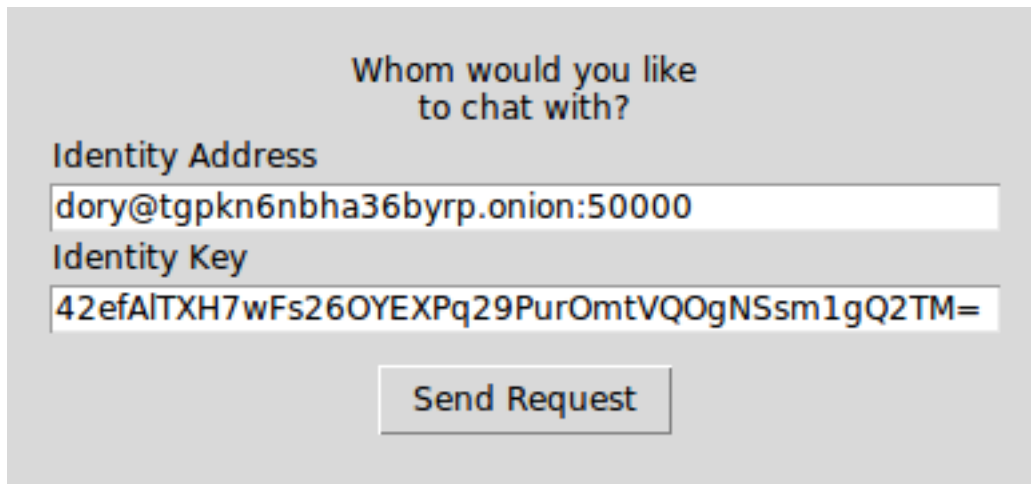


Fig. 1.2: Bootstrap window



Whom would you like
to chat with?

Identity Address

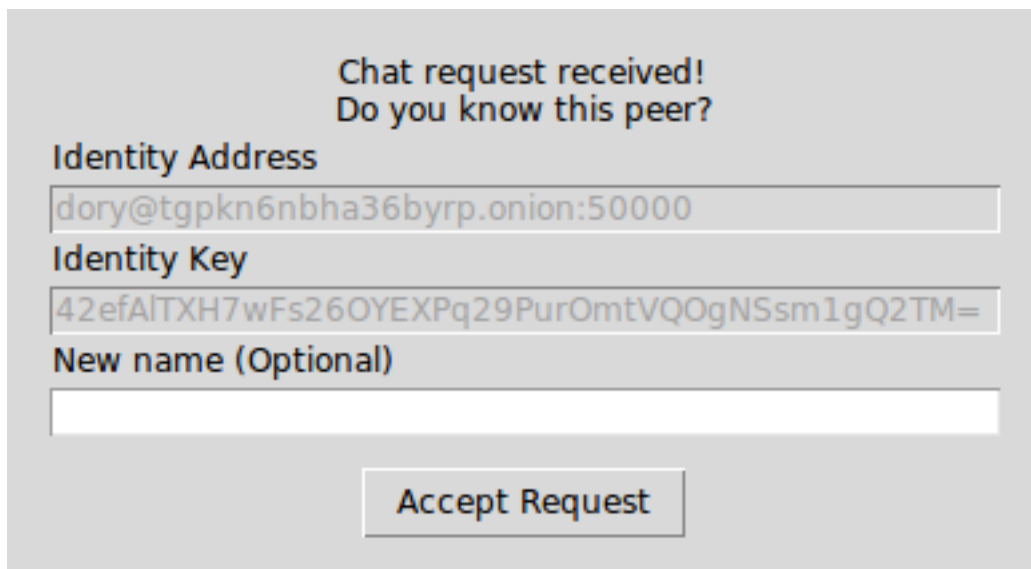
dory@tgpk6nbha36byrp.onion:50000

Identity Key

42efAlTXH7wFs26OYEXPq29PurOmtVQOgNSsm1gQ2TM=

Send Request

Fig. 1.3: Outbound request window



Chat request received!
Do you know this peer?

Identity Address

dory@tgpk6nbha36byrp.onion:50000

Identity Key

42efAlTXH7wFs26OYEXPq29PurOmtVQOgNSsm1gQ2TM=

New name (Optional)

Accept Request

Fig. 1.4: Inbound request window

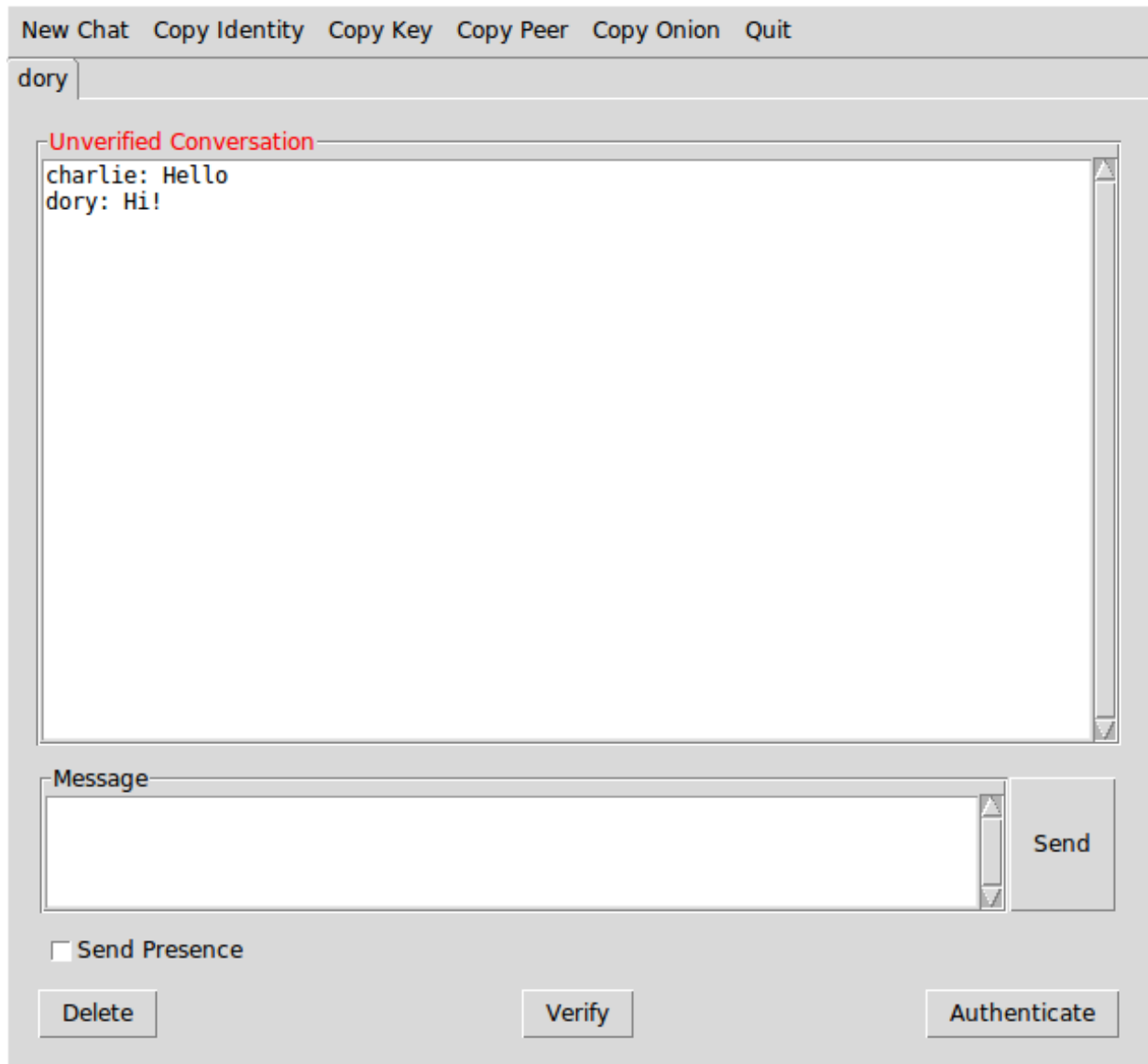


Fig. 1.5: Chat tab

Notifying Presence

If you wish to notify the peer whenever you go online or offline, check `Send Presence` and unMessage will start to send them notifications of these events.

Verifying

If you have some secure communication channel established with the other peer, ask them for their unMessage public identity key. Click `Verify` and enter the key:

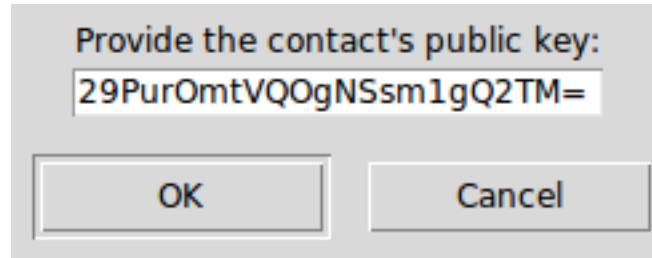


Fig. 1.6: Verification window

If the key matches, the peer will be verified and now you have established a verified and secure communication channel:

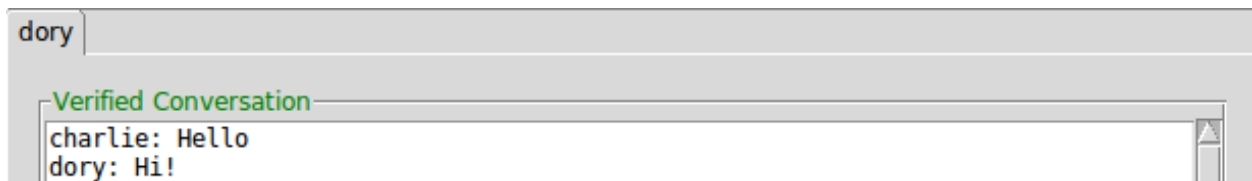


Fig. 1.7: Verified conversation

Authenticating

The authentication of a conversation works by prompting both peers for a secret (which was exchanged through some other secure channel) and if the secrets provided match, they are sure they are chatting with the right person. Click `Authenticate` and provide the secret:

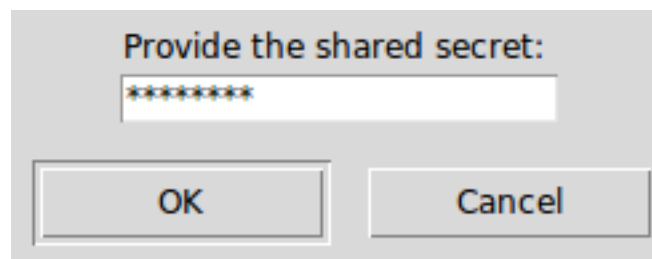


Fig. 1.8: Authentication window

An authentication session is created when the secrets are exchanged and is valid until one of the peers disconnect. When it happens, the conversation is not authenticated anymore and a new session must be initialized when the peers reconnect.

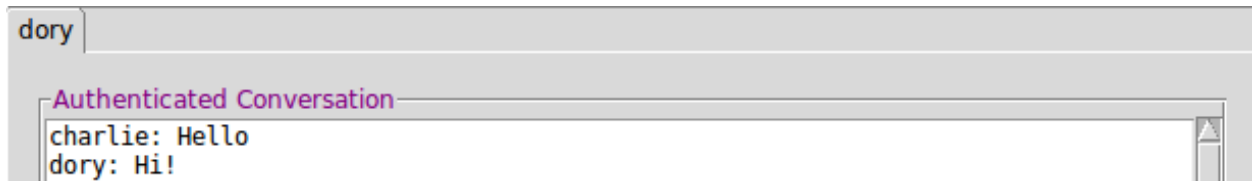


Fig. 1.9: Authenticated conversation

Assuming that one of the peers might be an attacker, this process is done with the [Socialist Millionaire Protocol](#) by comparing the secrets without actually disclosing them.

Authentication Levels

As noticed, unMessage conversations have three authentication levels:

1. Unverified Conversation
2. Verified Conversation
3. Authenticated Conversation

When the conversation is established, its level is **Unverified Conversation** because unMessage does not know if you are sure that the peer's identity key is actually theirs.

If you follow the [Verifying](#) section, the level changes to **Verified Conversation** and it persists for as long the **conversation** exists.

If you follow the [Authenticating](#) section, the level changes to **Authenticated Conversation** and it persists for as long the **session** exists. Once the **session** is over, the level drops to the identity key's verification level: **Unverified/Verified**.

Important: The **Authenticated** level is stronger than the **Verified** level because the former is a short term verification that lasts only until the peers disconnect, while the latter is long term that lasts until the conversation is deleted (manually, by the user). That means that with a short term verification you are able to authenticate the peer at that exact time, while a long term verification means that you authenticated the peer in the past, but is not aware of a compromise in the future.

This feature aims to increase unMessage's security by identifying an attack that is not covered by the scope of the *Double Ratchet Algorithm*: compromised keys.

1.3.4 Relaunching unMessage

unMessage remembers the last User Interface and Peer that you used. If you wish to use a shortcut, you may call:

```
unmessage
```

1.4 Command-line Interface (CLI)

To launch unMessage's *CLI*, pick any name you wish to use and call it with:

```
$ unmessage-cli -name <name>
```

Tor is launched and if this is the first time you use that name, your *Onion Service* and *Double Ratchet* keys are created and you are ready to receive and send requests to initialize conversations. unMessage displays this bootstrap process:

```
* unMessage

* Starting peer
* Configuring local server
* Running local server
* Configuring Tor
* Configuring Tor process
* Configuring Onion Service
* Running reactor
* 85%: Finishing handshake with first hop
* 90%: Establishing a Tor circuit

> □
```

Fig. 1.10: Bootstrap lines

After unMessage is launched, you can call `/help` to display all the commands the *CLI* responds to:

The `/peer`, `/onion` and `/key` commands can be used to copy information the other peers need to send you requests. You must share both your **identity address** and **key**:

```
bob@a7riwene46w3vqhp.onion RefK+9vx3GZpclb/On95iJlQnxqkUeq/JBYqK5gHFwo=
```

1.4.1 Sending Requests

Use the `/req-send` command to send a request, providing the **identity address** and **key** of the peer you wish to contact:

An **identity address** is provided in the format `<name>@<onion address>`, where the `<name>` is only a local identifier of the peer and you can pick any name you wish to call them.

1.4.2 Receiving Requests

Inbound requests are notified, with the information of the peer who sent the request:

As mentioned previously, peer names are local and when accepting a request you can pick another one to call them instead of using the one they sent.

```
* unMessage - bob@j1t6zabesvrhxvhee.onion:50000 v4kU6s+NuJW/Znbjz0AxoI9Gvl1XDS5ei0Tm6cE38E4=

> /help
/auth          authenticate a conversation with a shared secret
               args: <peer_name> <secret>
/convs          display existing conversations
/delete        delete conversation with a peer
               args: <peer_name>
/help          display commands that unMessage responds to
/identity      display your identity in the format <peer_name>@<onion_server>:<port>
/key           display your identity key
/msg           send message to a peer you maintain a conversation
               args: <peer_name> <message>
/onion         display your onion server
/peer          display your peer address and key
/pres-off      disable sending your presence to a peer at startup
               args: <peer_name>
/pres-on       enable sending your presence to a peer at startup
               args: <peer_name>
/quit          quit unMessage
/req-accept    accept a conversation request
               args: <peer_name>@<onion_server>:<port> [<new_peer_name>]
/req-send      send a conversation request
               args: <peer_name>@<onion_server>[:<port>] <identity_key>
/reqs-in       display inbound requests
/reqs-out      display outbound requests
/verify        verify a peer's identity key
               args: <peer_name> <identity_key>

>
```

Fig. 1.11: /help command

```
* unMessage - dory@tgpk6nbha36byrp.onion:50000 42efAlTXH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=

> /req-send charlie@j1t6zabesvrhxvhee.onion:50001 v4kU6s+NuJW/Znbjz0AxoI9Gvl1XDS5ei0Tm6cE38E4=
* Request sent: charlie@j1t6zabesvrhxvhee.onion:50001 has received your request

> □
```

Fig. 1.12: /req-send command

```
* unMessage - dory@tgpk6nbha36byrp.onion:50000 42efAlTXH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=

* Request received: charlie has sent you a request - accept using "/req-accept charlie@j1t6zabesvrhxvhee.onion:50001 [<new_peer_name>]"
> /req-accept charlie@j1t6zabesvrhxvhee.onion:50001
* Conversation established: You can now chat with charlie using "/msg charlie <message>"

> □
```

Fig. 1.13: /req-accept command

1.4.3 Chatting

unMessage displays each peer you have a conversation with by calling the `/convs` command.

```
* unMessage - dory@tgpk6nbha36byrp.onion:50000 42efAlTXH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=  
  
* Conversations:  
  charlie@jt6zabesvrhxvhee.onion:50001 v4kU6s+NuJW/Znbjz0AxoI9Gv1lXDS5ei0Tm6cE38E4=  
  
> 
```

Fig. 1.14: `/convs` command

To send a message to a peer, use the `/msg` command:

```
* unMessage - dory@tgpk6nbha36byrp.onion:50000 42efAlTXH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=  
  
> /msg charlie Hi!  
charlie< Hi!  
charlie< 
```

Fig. 1.15: `/msg` command

Notifying Presence

If you wish to notify the peer whenever you go online or offline, use the `/pres-on` command and unMessage will start to send them notifications of these events:

```
* unMessage - dory@tgpk6nbha36byrp.onion:50000 42efAlTXH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=  
  
> /pres-on charlie  
* You will start sending your presence to charlie  
> 
```

Fig. 1.16: `/pres-on` command

To disable, use the `/pres-off` command.

Verifying

If you have some secure communication channel established with the other peer, ask them for their unMessage public identity key. Use the `/verify` command and enter the key:

```
* unMessage - dory@tgpk6nbha36byrp.onion:50000 42efALTxH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=
charlie< /verify charlie v4kU6s+NujW/Znbjz0AxoI9Gv1lXD55ei0Tm6cE38E4=
* charlie's key has been verified.
charlie< 
```

Fig. 1.17: `/verify` command

If the key matches, the peer will be verified and now you have established a verified and secure communication channel.

Authenticating

The authentication of a conversation works by prompting both peers for a secret (which was exchanged through some other secure channel) and if the secrets provided match, they are sure they are chatting with the right person. Call the `/auth` command and provide the secret:

```
* unMessage - dory@tgpk6nbha36byrp.onion:50000 42efALTxH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=
* Authentication started: charlie wishes to authenticate - advance using "/auth charlie <secret>"
> /auth charlie axolotl
* Authentication successful: Your conversation with charlie is authenticated!
> 
```

Fig. 1.18: `/auth` command

An authentication session is created when the secrets are exchanged and is valid until one of the peers disconnect. When it happens, the conversation is not authenticated anymore and a new session must be initialized when the peers reconnect.

Assuming that one of the peers might be an attacker, this process is done with the [Socialist Millionaire Protocol](#) by comparing the secrets without actually disclosing them.

Authentication Levels

As noticed, the names of the peers are colored based on the conversation authentication levels:

1. Unverified Conversation (red)
2. Verified Conversation (green)
3. Authenticated Conversation (cyan)

When the conversation is established, its level is **Unverified Conversation** because unMessage does not know if you are sure that the peer's identity key is actually theirs.

If you follow the [Verifying](#) section, the level changes to **Verified Conversation** and it persists for as long the **conversation** exists.

If you follow the [Authenticating](#) section, the level changes to **Authenticated Conversation** and it persists for as long the **session** exists. Once the **session** is over, the level drops to the identity key's verification level: **Unverified/Verified**.

Important: The **Authenticated** level is stronger than the **Verified** level because the former is a short term verification that lasts only until the peers disconnect, while the latter is long term that lasts until the conversation is deleted (manually, by the user). That means that with a short term verification you are able to authenticate the peer at that exact time, while a long term verification means that you authenticated the peer in the past, but is not aware of a compromise in the future.

This feature aims to increase unMessage's security by identifying an attack that is not covered by the scope of the *Double Ratchet Algorithm*: compromised keys.

1.4.4 Relaunching unMessage

unMessage remembers the last User Interface and Peer that you used. If you wish to use a shortcut, you may call:

```
unmessage
```

Note: unMessage's CLI is inspired by [xmpp-client](#).

1.5 unMessage Protocol

The unMessage protocol is based on the [Double Ratchet Algorithm](#) to establish conversations and exchange messages privately and anonymously.

Note: unMessage uses [Tor Onion Services](#) to anonymously connect peers as we believe that it is the best transport for this kind of application, but other approaches such as posting the packets to a public mailing list should also work (as long as the packets are anonymously posted).

In the *Double Ratchet Algorithm*, a **secret key** must be agreed on to derive all the other keys involved in the conversation. The **secret key** used by unMessage is generated with the [Triple Diffie-Hellman Key Agreement](#), using one party's **public identity and handshake keys**, and another's **private identity and handshake keys**.

Each party must have its mode assigned to as either **Alice** or **Bob**. The one who starts the initialization is **Bob** and can send messages right after the **secret key** is generated. As part of the initialization, **Bob** must send his **public ratchet key** to **Alice** so that she is able to start the [Diffie-Hellman ratcheting](#) and also send messages immediately.

unMessage conversations have the following stages:

1. Request sent
2. Request accepted
3. Conversation established

In order to send requests, both parties must launch unMessage to generate their *Onion Service* and *Double Ratchet* keypairs. unMessage is a **serverless** application, so a peer who wishes to receive requests must send/publish their *Onion Service* address and *Double Ratchet* public identity key through some other communication channel.

unMessage assigns **Bob** to the one who sends a request and **Alice** to the one who receives it.

Important: In the following sections, the **shared request key** and **conversation ID** are described as the direct input of hash and encryption functions for simplicity. In fact, these keys are input of a *Key Derivation Function (KDF)* along with its respective *salt*, and the output keys of the *KDF* that are actually used by such functions.

1.5.1 Stage 1: Request sent

A **request keypair** is generated by **Bob's** unMessage to derive a *Diffie-Hellman* **shared request key** using the **private request key** and **Alice's public identity key**. The **shared request key**, is used to encrypt the following information needed by **Alice** to initialize a conversation with **Bob**:

- Bob's identity address
- Bob's identity public key
- Bob's handshake public key
- Bob's ratchet public key

This set composes the **handshake packet**, which after encrypted is used to compose the **request packet**:

- IV
- hash(IV + Alice's public identity key + shared request key)
- keyed_hash(shared request key, encrypted handshake packet)
- public request key
- encrypted handshake packet

The packet is then sent to **Alice's Onion Address** and **Stage 1** is completed.

Important: The **handshake packet** should be signed by the *Onion Service* and *Double Ratchet* keys so that a peer cannot advertise keys they do not own. This will be implemented in a future version of unMessage.

1.5.2 Stage 2: Request accepted

After receiving the **request packet**, **Alice's** unMessage derives the **shared request key** using **Alice's private identity key** and the **public request key**. The **shared request key** is hashed with the **IV** and the **handshake packet** to make sure that is indeed an unMessage **request packet** and the **handshake packet** can be decrypted. **Alice** is notified that the request was received from **Bob** and accepts it to initialize the *Double Ratchet* conversation.

Bob's public identity and handshake keys sent in the **handshake packet** are used to generate the *Double Ratchet* **secret key** with **Alice's private identity and handshake keys** (the former was generated when unMessage was launched by the first time and the latter when the request was accepted, to be used for this specific conversation). The *Double Ratchet* conversation is finally initialized using the **secret key** and **Bob's public ratchet key** (also sent in the **handshake packet**). At this point, **Stage 2** is completed and **Alice** can start sending encrypted messages. However, as **Bob** does not have **Alice's public handshake key**, it is encrypted (using the **shared request key**) and sent along with the unMessage **reply packet**:

- IV
- hash(IV + Bob's public identity key + shared request key)
- keyed_hash(shared request key, encrypted handshake key + encrypted payload)
- Alice's encrypted public handshake key
- encrypted payload

1.5.3 Stage 3: Conversation established

When messages from **Alice** are received, **Bob's** unMessage hashes the **shared request key** with the **IV** and **Alice's encrypted public handshake key** concatenated with the **encrypted payload** to make sure that is indeed an unMessage **packet** from **Alice**, and her **public handshake key** can be decrypted. **Bob** now can also generate the **secret key** with his **private identity and handshake keys**, and **Alice's public identity and handshake keys**. With his part of the conversation initialized, he can start sending unMessage **regular packets**:

- IV
- hash(IV + Alice's public identity key + conversation ID)
- keyed_hash(conversation ID, encrypted payload)
- encrypted payload

Stage 3 is completed when **Alice** receives a **regular packet** from **Bob**, which means that he was able to initialize the conversation with her **public handshake key** and there is no need to send **reply packets** anymore, so her unMessage also starts sending **regular packets**.

1.5.4 Identifying conversations

All of the identifying information of an unMessage packet is encrypted so that an attacker who intercepts it cannot tell who are the receiver and sender.

When a packet is received, unMessage assumes it is a **regular packet** and attempts to use all of the peer's **conversation IDs** to derive the **IV hash**. If the hash matches the packet's **IV hash**, unMessage identifies the sender and is able to decrypt the **payload** (after verifying its integrity). If the **IV hash** does not match, unMessage assumes the packet is a **request packet** and derives a **shared request key** using the **public request key** from the packet and the peer's **public identity key**. unMessage attempts to use the **shared request key** and the **IV** to derive a hash that matches the packet's **IV hash**. If it matches, unMessage checks the integrity of the rest of the packet and processes the request as described in **Stage 2**.

When unMessage fails to identify or check the integrity of packets, they are ignored.

Note: The **IV hash** also uses the receiver's public identity key as part of the hash so that, for example, Alice can tell the difference between messages she sent to Bob and messages she received from Bob.

The **IV hash** is another implementation of an [hSub](#).

2.1 Changelog

2.1.1 unMessage 0.1.0, released 2017-01-22

- Initial commit

2.2 Feedback

Please join us on **#unMessage:anemone.me** or **#anemone:anemone.me** with [Matrix](#), **#anemone** at [OFTC](#), or use the [GitHub issue tracker](#) to leave suggestions, bug reports, complaints or anything you feel will contribute to this application.