

---

# **unMessage Documentation**

***Release 0.1.0***

**Anemone Labs**

**Sep 24, 2017**



---

## Contents

---

<b>1</b>	<b>What is it?</b>	<b>1</b>
1.1	Overview . . . . .	1
1.2	Installation . . . . .	1
1.3	Usage . . . . .	3
1.4	Persistence . . . . .	3
1.5	Graphical User Interface (GUI) . . . . .	3
1.6	Command-line Interface (CLI) . . . . .	9
1.7	unMessage Protocol . . . . .	15
<b>2</b>	<b>Other</b>	<b>23</b>
2.1	Changelog . . . . .	23
2.2	Feedback . . . . .	24



---

## What is it?

---

### 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.

### 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)

### Installation

unMessage's installation is done in three steps:

1. Install requirements
2. Use a virtual environment
3. Install unMessage

## Requirements

Install the following requirements via package manager:

```
$ # If using Debian/Ubuntu
$ sudo apt-get install build-essential gcc libffi-dev libopus0 \
  libsodium-dev libssl-dev portaudio19-dev python-dev python-tk

$ # If using Fedora
$ sudo dnf install gcc libffi-devel libsodium-devel \
  openssl-devel opus portaudio-devel python-devel \
  redhat-rpm-config tkinter
```

If you have **tor** installed, make sure its version is at least 0.2.7.1:

```
$ tor --version
```

If you must update it or do not have it installed, check the version provided by the package manager:

```
$ # If using Debian/Ubuntu
$ apt-cache show tor

$ # If using Fedora
$ dnf info tor
```

If the version to be provided is not at least 0.2.7.1, you will have to [set up Tor's package repository](#). Once you have a repository which can provide an updated **tor**, install it:

```
$ # If using Debian/Ubuntu
$ sudo apt-get install tor

$ # If using Fedora
$ sudo dnf install tor
```

## Using a Virtual Environment

Install **virtualenv**, **pip** and **setuptools**:

```
$ # If using Debian/Ubuntu
$ sudo apt-get install python-virtualenv

$ # If using Fedora
$ sudo dnf install python-virtualenv
```

Use a *virtual environment*:

```
$ virtualenv ~/unmessage-env      # create
$ . ~/unmessage-env/bin/activate # activate
(unmessage-env)$                 # prompt shows which environment is active
```

Update **setuptools**, **pip** and **virtualenv**:

```
(unmessage-env)$ pip install --upgrade setuptools
(unmessage-env)$ pip install --upgrade pip
(unmessage-env)$ pip install --upgrade virtualenv
```

Make sure that the update installs at least *pip* 8 and *setuptools* 19.4.

## Installing

Finally, install unMessage:

```
(unmessage-env)$ pip install unmessage
```

Launch unMessage with any of the commands:

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

Make sure to activate the *virtual environment* whenever you wish to use unMessage:

```
$ . ~/unmessage-env/bin/activate
```

As well as deactivate it when you are done:

```
(unmessage-env)$ deactivate
```

## Updating

*pip* can also be used to update unMessage:

```
(unmessage-env)$ pip install --upgrade unmessage
```

## Usage

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

## Persistence

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

## 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**:

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



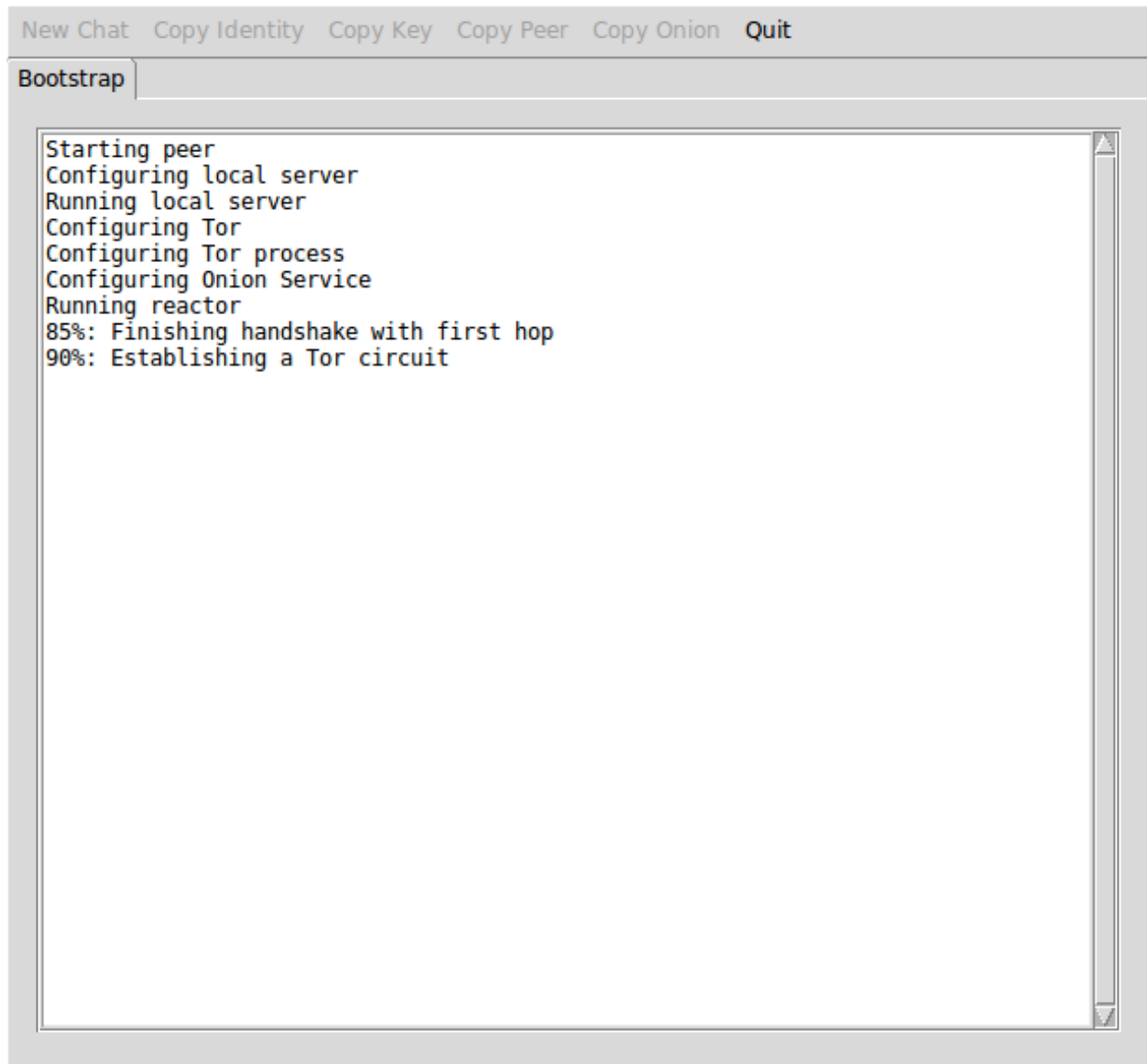


Fig. 1.2: Bootstrap window

```
charlie@jt6zabesvrhxee.onion:50001 v4kU6s+NuJW/Znbjz0AxoI9Gv1lXDS5eiOTm6cE38E4=
```

## 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:

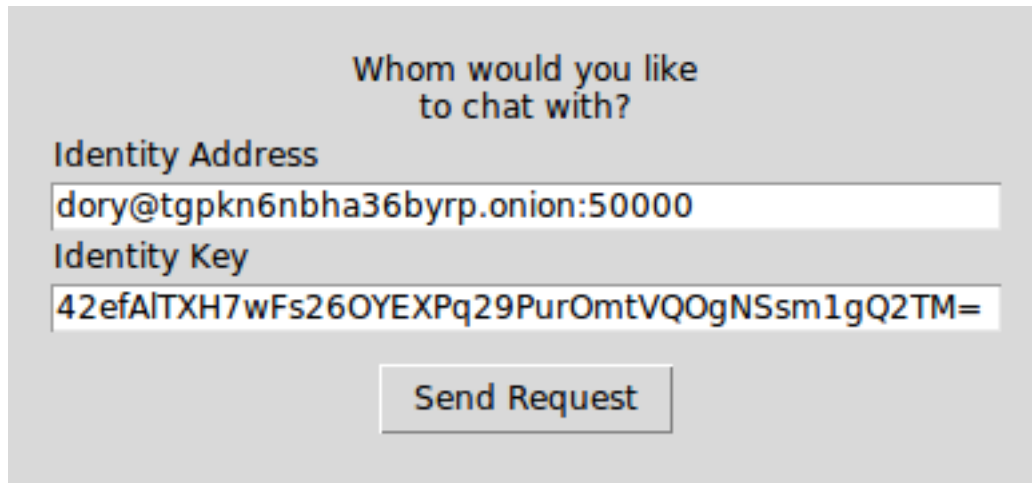


Fig. 1.3: Outbound request window

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.

## 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.

## 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.

### 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:

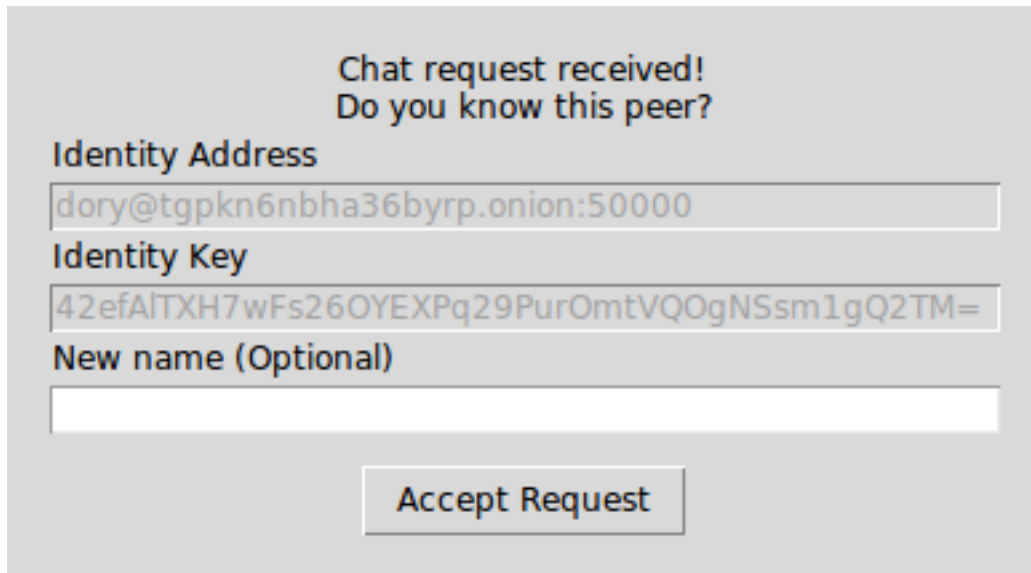


Fig. 1.4: Inbound request window

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. Click `Authenticate` and provide the secret:

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, 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**.

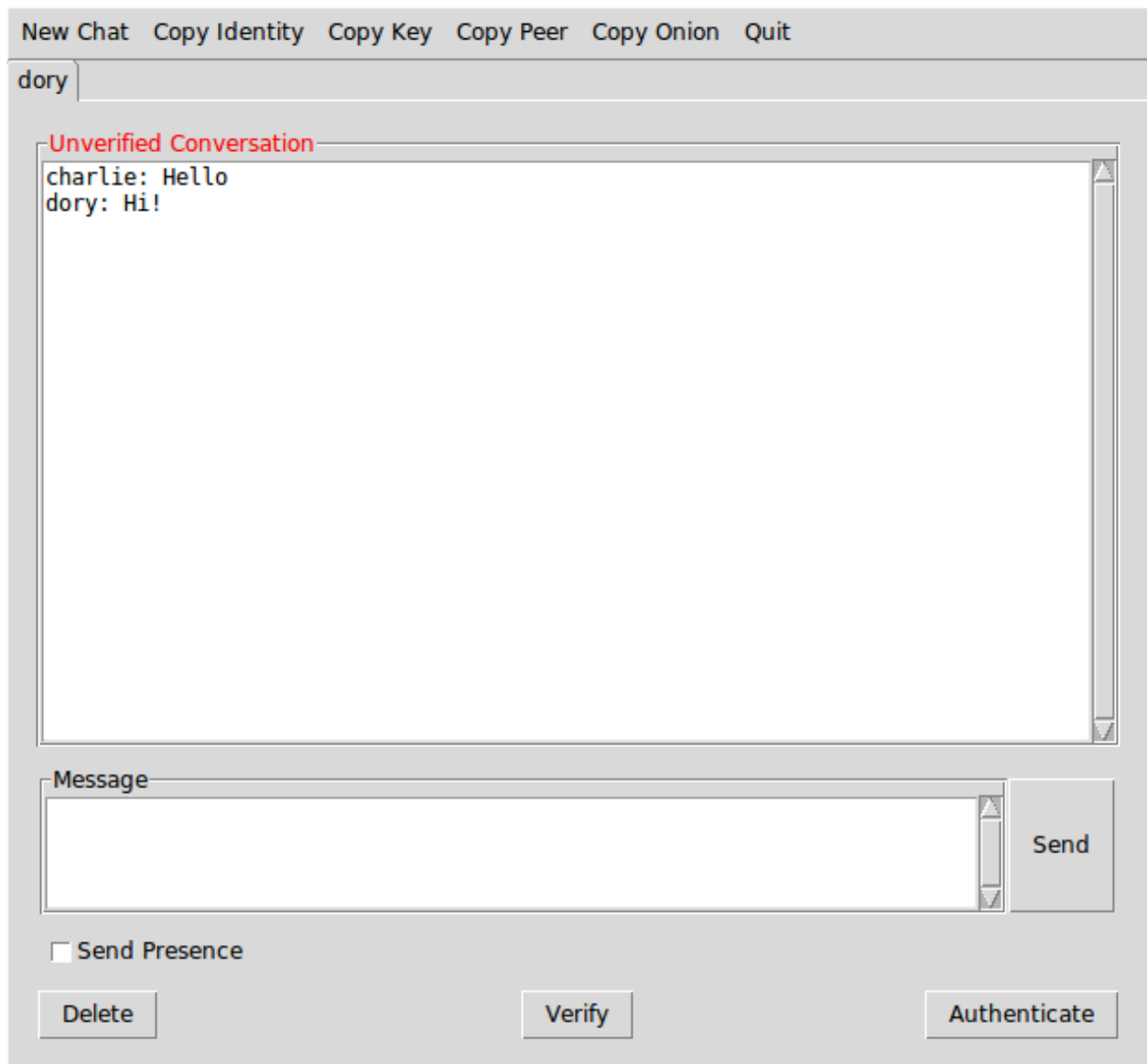


Fig. 1.5: Chat tab

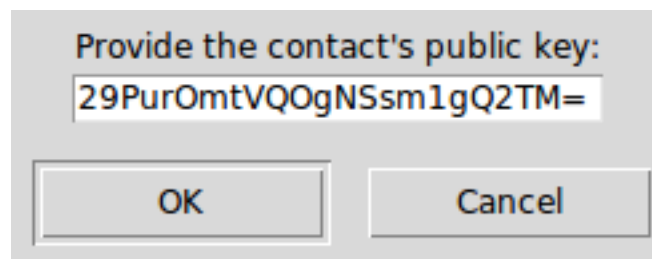


Fig. 1.6: Verification window

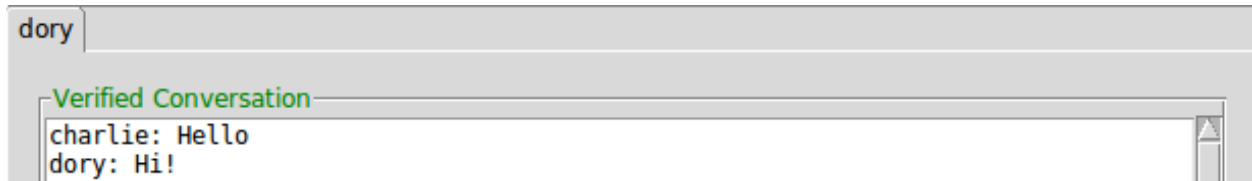


Fig. 1.7: Verified conversation

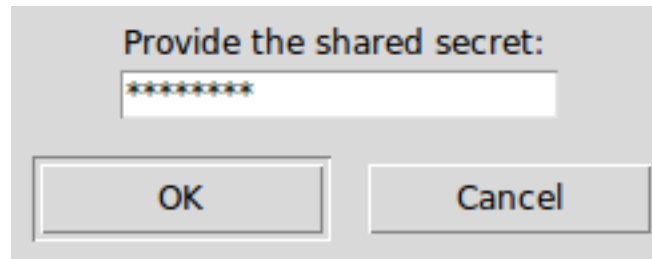


Fig. 1.8: Authentication window

**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.

## Relaunching unMessage

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

```
unmessage
```

## Command-line Interface (CLI)

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

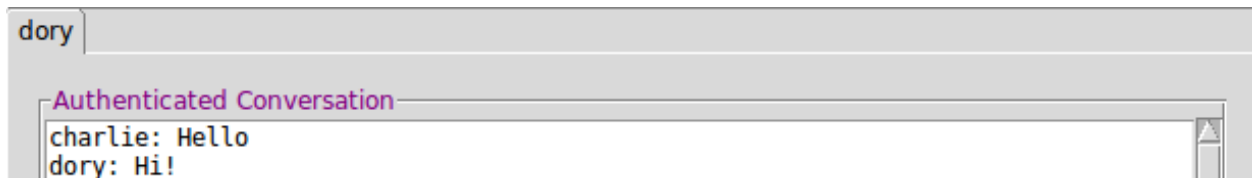


Fig. 1.9: Authenticated conversation

```
$ 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:

A terminal window with a black background and light blue text. The text shows the output of the 'unMessage' command. It starts with '\* unMessage' followed by several status messages: '\* Starting peer', '\* Configuring local server', '\* Running local server', '\* Configuring Tor', '\* Configuring Tor process', '\* Configuring Onion Service', '\* Running reactor', '\* 85%: Finishing handshake with first hop', and '\* 90%: Establishing a Tor circuit'. At the bottom left, there is a prompt '>' followed by a small square cursor.

```
* 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=
```

## 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.

## 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@jt6zabesvrhxvhee.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@tgpkn6nbha36byrp.onion:50000 42efAlTXH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=

> /req-send charlie@jt6zabesvrhxvhee.onion:50001 v4kU6s+NuJW/Znbjz0AxoI9Gvl1XDS5ei0Tm6cE38E4=
* Request sent: charlie@jt6zabesvrhxvhee.onion:50001 has received your request

> □
```

Fig. 1.12: /req-send command

```
* unMessage - dory@tgpkn6nbha36byrp.onion:50000 42efAlTXH7wFs260YEXPq29Pur0mtVQ0gNSsm1gQ2TM=

* Request received: charlie has sent you a request - accept using "/req-accept charlie@jt6zabesvrhxvhee.onion:50001 [<new_peer_name>]"
> /req-accept charlie@jt6zabesvrhxvhee.onion:50001
* Conversation established: You can now chat with charlie using "/msg charlie <message>"

> □
```

Fig. 1.13: /req-accept command

## Chatting

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

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

* Conversations:
  charlie@j6t6zabesvrhvxvhee.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

## Transmitting Files

unMessage also supports file transfers. Sending a request to transmit a file can be done with the `/file-send` command:

```
/file-send charlie ~/file.txt
```

The other party will receive a notification with the file name, size and checksum. It can be authorized to be transmitted with the `/file-accept` command:

```
/file-accept dory DiEjQOChrEorC0iPxrdNenBhiITaobehz5sQSkNnWIY=
```

The file will be saved by default as `~/.config/unMessage/<your-peer>/conversations/<other-peer>/file-transfer/<original-file-name>`. The command also supports an optional argument for the path to save the file instead of using the default one:

```
/file-accept dory DiEjQOChrEorC0iPxrdNenBhiITaobehz5sQSkNnWIY= ~/dory.txt
```



Once the initiator receives the confirmation, the file transfer is finally initialized and both parties are notified when it is complete.

## 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/Znbjz0AxoI9Gvl1XDS5ei0Tm6cE38E4=
* 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:

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.

```
* unMessage - dory@tgpgkn6nbha36byrp.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

## 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.

---

## 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](#).

---

## unMessage Protocol

This section describes the logic for sending/accepting requests and exchanging messages in *Establishing Conversations*, as well as the packets used in each of those stages in *Packet Formats*.

### Establishing Conversations

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.

---

#### 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
- $\text{hash}(\text{IV} + \text{Alice's public identity key} + \text{shared request key})$
- $\text{keyed\_hash}(\text{shared request key}, \text{encrypted handshake packet})$
- 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.

---

## 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
- $\text{hash}(\text{IV} + \text{Bob's public identity key} + \text{shared request key})$
- $\text{keyed\_hash}(\text{shared request key}, \text{encrypted handshake key} + \text{encrypted payload})$
- Alice's encrypted public handshake key
- encrypted payload

## 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
- $\text{hash}(\text{IV} + \text{Alice's public identity key} + \text{conversation ID})$
- $\text{keyed\_hash}(\text{conversation ID}, \text{encrypted payload})$
- 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**.

## 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](#).

---

## Packet Formats

unMessage's conversations have three stages, each using a different packet format:

- **Request:** contains Bob's name, address and keys (identity, handshake and ratchet)
- **Reply:** contains Alice's key (handshake) and optionally an encrypted element
- **Regular:** contains an encrypted element

---

**Note:** **Elements** are the plaintext of the information exchanged in unMessage's conversations, which are wrapped by Double Ratchet's encryption and added to reply/regular packets for transmission. (e.g., *presence notifications, text messages, authentication buffers*)

---

The following sections summarize what each packet is used for, their exact contents and their size in *bytes*.

(In the following diagrams, data surrounded by === is encrypted)

### Request Packet

To notify **Alice** that **Bob** wishes to establish a conversation with her, he must send all the information she needs to complete this process. The information is sent in a **request packet**:

```
+-----+
| Request packet (240 + address) |
+-----+
| IV (8) |
| IV Hash (32) |
| Keyed hash (32) |
| Public request key (32) |
| |
| +-----+
| | Encrypted handshake packet (136 + address) | |
| +-----+
+-----+
```

```

| | Nonce (24) | | | |
| | MAC (16) | |
| | +=====+ | |
| | | Identity address | | |
| | | Public identity key (32) | | |
| | | Public handshake key (32) | | |
| | | Public ratchet key (32) | | |
| | | +=====+ | |
| | +-----+ | |
+-----+

```

The **request key** is used to derive a **shared request key** with **Alice's** identity key in order to encrypt **Bob's** information so that only the ones in possession of the private **request** or **identity** keys are able to read who sent the request.

## Reply Packet

Once **Alice** accepts the request, she is able to send encrypted elements to **Bob**, who sent all information required by her to initialize a conversation. However, as **Bob** needs her **handshake key**, she adds it before the payload of the message, in case an element should also be included. This information is sent in a **reply packet**:

```

+-----+
| Reply packet (192 + 72 + payload) |
+-----+
| IV (8) |
| IV Hash (32) |
| Keyed hash (32) |
|
| +-----+
| | Encrypted public handshake key (72) | |
| +-----+
| | Nonce (24) | | | |
| | MAC (16) | |
| | +=====+ | |
| | | Public handshake key (32) | | |
| | +=====+ | |
| +-----+
|
| +-----+
| | Encrypted payload (120 + payload) | |
| +-----+
| | +-----+
| | | Double Ratchet header (80) | | |
| | +-----+
| | | Nonce (24) | | | | |
| | | MAC (16) | | |
| | | +=====+ | | |
| | | | Ns (3) | | | |
| | | | PNs (3) | | | |
| | | | DHRs (32) | | | |
| | | +=====+ | | |
| | | Padding (2) | | |
| | +-----+
| |
| | +-----+
| | | Double Ratchet payload (40 + payload) | | |
| | +-----+
| | | Nonce (24) | | |

```

```

| | | MAC (16) | | |
| | | +=====+ | | |
| | | Payload | | |
| | | +=====+ | | |
| | +-----+ | | |
| +-----+ | | |
+-----+

```

In order to send multiple messages to **Bob** (which might be delivered out of order), **Alice** must continue to send her **handshake key** until **Bob** replies (signaling that he was able to establish a conversation as well). To prevent **reply packets** from being linked by leaking the **handshake key**, it is encrypted using the **shared request key** used in the encryption of the **request packet** sent by **Bob**.

### Regular Packet

Once both peers have initialized their sides of the conversation, there is no need for **Alice** to send the **handshake key** anymore. The only content subsequent exchanges transmit are their payloads. This information is sent in a **regular packet**:

```

+-----+
| Regular packet (192 + payload) |
+-----+
| IV (8) |
| IV Hash (32) |
| Keyed hash (32) |
|
| +-----+
| | Encrypted payload (120 + payload) | |
| +-----+
| | +-----+
| | Double Ratchet header (80) | |
| | +-----+
| | Nonce (24) | |
| | MAC (16) | |
| | +=====+
| | | Ns (3) | |
| | | PNs (3) | |
| | | DHRs (32) | |
| | +=====+
| | Padding (2) | |
| | +-----+
| |
| | +-----+
| | Double Ratchet payload (40 + payload) | |
| | +-----+
| | Nonce (24) | |
| | MAC (16) | |
| | +=====+
| | | Payload | |
| | +=====+
| | +-----+
| +-----+
+-----+

```

**Important:** Despite the fact that each packet's contents look like random information, in the current version of

unMessage all of them have a different size. In the future, all packets should be padded to a fixed size in order to achieve indistinguishability.

---

## Threat Model

unMessage is characterized by the packets it creates and processes, and the transport used to transmit such packets between its peers. Tor Onion Services is the current supported solution used to connect peers, but as unMessage employs an application protocol that manages its own packets, it would be possible to allow the use of other transports as long as such packets are transmitted from one peer to another, anonymously.

It is expected that the transport connecting the peers conceals their real identity, location and path of transmissions from each other as well as from an external adversary observing the network that is not as powerful as a Global Passive Adversary. From this perspective, unMessage is susceptible to the same security vulnerabilities as the transport in use.

Although unMessage expects that information to be anonymously *exchanged* between the peers, it does not require anything beyond that because by default its packet format provides:

- Integrity
- Authenticity
- Confidentiality
- Anonymity

As conversations are established between peers with Double Ratchet sessions, they also benefit from the properties of:

- Forward secrecy
- Future secrecy
- Deniability

## Adversary Capabilities

From the application's perspective, taking into account the local server availability, packet creation and packet processing, we assume the following capabilities from an adversary:

1. An adversary is unable to break the cryptographic primitives used by unMessage.
2. An adversary is able to observe, intercept, replay and modify all packets exchanged by the peers.
3. An adversary is able to send requests and malformed packets to a peer whose unMessage address and public identity key has been acknowledged by them.
4. An adversary is unable to perform an attack by making multiple connections or sending multiple requests to a peer whose unMessage address has been acknowledged by them, making that peer unavailable to others.
5. An adversary is unable to send malformed/malicious elements to a peer who accepted their request and therefore has established a conversation with.
6. An adversary is unable to compromise a peer's private identity key to impersonate them in current and future conversations.
7. An adversary is unable to compromise a peer's private identity key to decrypt any of the requests they received/accepted.



## Possible Attacks

Some limitations to the adversary's capabilities had to be imposed due to the current implementation of unMessage, which does not yet prevent some of the attacks mentioned in *Adversary Capabilities*:

- unMessage maps an Onion Service to a local server that accepts connections and receives netstrings to be parsed as unMessage packets. This behavior allows an adversary who has knowledge of a peer's Onion Service address to perform the attack mentioned in item 4, by either making the Onion Service inaccessible in the network or overloading the unMessage instance.
- Although unMessage validates the format of the packets it expects to receive and an adversary cannot make any modifications due to the integrity checks, once a packet is decrypted after being validated by passing such checks, there is not yet a mechanism validating the conversation elements (i.e., the plaintext of reply/regular packets) and the attack mentioned in item 5 is possible to be performed.
- The attack mentioned in item 6 can be mitigated as unMessage provides an authentication feature. As long as the users have securely agreed on a secret that is not known by the adversary, the party who the impersonator is communicating with can use it to initiate the authentication process and detect the attack. Even though any user can initiate the authentication at any time after a conversation is established, it is up to them to properly handle the secret and regularly authenticate themselves - or at least do so under any suspicion. Even if users take such actions, unMessage is only able to detect instead of prevent it.
- All the peer information required to send/receive conversation requests is encrypted with a shared secret derived from a Diffie-Hellman key exchange using Bob's request key and Alice's identity key. The request key is ephemeral and is disposed once the conversation is established, but the identity key is not. For that reason, the attack mentioned in item 7 is possible to be performed but not prevented by unMessage.



### Changelog

#### **unMessage 0.2.0, released 2017-05-12**

- Support multiple conversation managers
- Support voice conversations with unTalk
- Support connecting to the system's Tor
- Use Ephemeral Onion Services
- Allow customization of the local server's network interface
- Fix bug which allowed any command on the CLI before bootstrapping
- Fix bug which ignored element packets longer than 4 lines and consequently any multiline message

#### **unMessage 0.1.1, released 2017-02-10**

- Improve ports handling
- Improve handling of unusual packets

#### **unMessage 0.1.0, released 2017-01-22**

- Initial commit

## 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.