Magic-Wormhole Documentation

Release 0.13.0+26.g88f7548.dirty

Brian Warner

Sep 20, 2023

Contents:

1 Welcome 1
- 1.1 Example 1
- 1.2 Installation 2
- 1.3 Motivation 3
- 1.4 Design 4
- 1.5 Timing 4
- 1.6 Relays 4
- 1.7 CLI tool 5
- 1.8 Library 5
- 1.9 Development 5
- 1.10 License, Compatibility 6

2 The Magic-Wormhole Ecosystem 7
- 2.1 Documentation 7
- 2.2 Protocols Overview 7
- 2.3 Implementations and Support 8
- 2.4 Features Supported by Implementations 8
- 2.5 End User / Client Applications 9
- 2.6 Integrations 9

3 Tor Support in Magic-Wormhole 11
- 3.1 Usage 11
- 3.2 Other Ways To Reach Tor 12
- 3.3 .onion servers 12

4 Protocol/API/Library Introduction 13
- 4.1 The Magic-Wormhole Protocol 13
- 4.2 The `wormhole` API 14

5 The Magic-Wormhole API 15
- 5.1 Modes 16
- 5.2 Application Identifier 16
- 5.3 Mailbox Servers 16
- 5.4 Wormhole Parameters 17
- 5.5 Code Management 17
- 5.6 Offline Codes 19
- 5.7 Welcome Messages 19

5.8 Verifier 20
5.9 Events 21
5.10 Sending Data 22
5.11 Closing 22
5.12 Serialization 22
5.13 Dilation 23
5.14 Bytes, Strings, Unicode, and Python 3 25
5.15 Full API list 25

6 Transit Protocol **27**
6.1 Roles 27
6.2 Records 28
6.3 Handshake 28
6.4 Relay 29
6.5 API 29

7 Mailbox Server Protocol **31**
7.1 Concepts 31
7.2 Application IDs 31
7.3 WebSocket Transport 31
7.4 Connection-Specific (Client-to-Server) Messages 32
7.5 Nameplates 32
7.6 Mailboxes 33
7.7 All Message Types 34
7.8 Persistence 34

8 Client-to-Client Protocol **37**

9 File-Transfer Protocol **39**
9.1 Sender 39
9.2 Recipient 40
9.3 Transit 40
9.4 Future Extensions 41

10 Known Vulnerabilities **43**
10.1 Low-probability Man-In-The-Middle Attacks 43
10.2 DoS Attack on the Mailbox Server 43

11 Journaled Mode **45**
11.1 Requirements 45
11.2 Wormhole Support 46

12 Indices and tables **49**

CHAPTER 1

Welcome

Get things from one computer to another, safely.

This package provides a library and a command-line tool named `wormhole`, which makes it possible to get arbitrary-sized files and directories (or short pieces of text) from one computer to another. The two endpoints are identified by using identical "wormhole codes": in general, the sending machine generates and displays the code, which must then be typed into the receiving machine.

The codes are short and human-pronounceable, using a phonetically-distinct wordlist. The receiving side offers tab-completion on the codewords, so usually only a few characters must be typed. Wormhole codes are single-use and do not need to be memorized.

- PyCon 2016 presentation: Slides, Video

As of now (2023) the magic-wormhole protocol has several client implementations; see the "Ecosystem" section.

1.1 Example

Sender:

```
% wormhole send README.md
Sending 7924 byte file named 'README.md'
On the other computer, please run: wormhole receive
Wormhole code is: 7-crossover-clockwork

Sending (<-10.0.1.43:58988)..
100%|=========================| 7.92K/7.92K [00:00<00:00, 6.02MB/s]
File sent.. waiting for confirmation
Confirmation received. Transfer complete.
```

Receiver:

```
% wormhole receive
Enter receive wormhole code: 7-crossover-clockwork
```

(continues on next page)

(continued from previous page)

```
Receiving file (7924 bytes) into: README.md
ok? (y/n): y
Receiving (->tcp:10.0.1.43:58986)..
100%|===========================| 7.92K/7.92K [00:00<00:00, 120KB/s]
Received file written to README.md
```

1.2 Installation

The easiest way to install magic-wormhole is to use a packaged version from your operating system. If there is none, or you want to participate in development, you can install from source.

1.2.1 MacOS / OS-X

Install Homebrew, then run `brew install magic-wormhole`.

1.2.2 Linux (Debian/Ubuntu)

Magic-wormhole is available with `apt` in Debian 9 "stretch", Ubuntu 17.04 "zesty", and later versions:

```
$ sudo apt install magic-wormhole
```

1.2.3 Linux (Fedora)

Note: magic-wormhole was removed from Fedora starting in Fedora 37. So this command will only work on Fedora 36 and earlier.

```
$ sudo dnf install magic-wormhole
```

1.2.4 Linux (openSUSE)

```
$ sudo zypper install python-magic-wormhole
```

1.2.5 Linux (Snap package)

Many linux distributions (including Ubuntu) can install "Snap" packages. Magic-wormhole is available through a third-party package (published by the "snapcrafters" group):

```
$ sudo snap install wormhole
```

1.2.6 Windows

Chocolatey

```
$ choco install magic-wormhole
```

The binaries for Windows are provided from this project: https://github.com/aquacash5/magic-wormhole-exe

1.2.7 Install from Source

Magic-wormhole is a Python package, and can be installed in the usual ways. The basic idea is to do `pip install magic-wormhole`, however to avoid modifying the system's python libraries, you probably want to put it into a "user" environment (putting the `wormhole` executable in `~/.local/bin/wormhole`) like this:

```
pip install --user magic-wormhole
```

or put it into a virtualenv, like this:

```
virtualenv venv
source venv/bin/activate
pip install magic-wormhole
```

You can then run `venv/bin/wormhole` without first activating the virtualenv, so e.g. you could make a symlink from `~/bin/wormhole` to `.../path/to/venv/bin/wormhole`, and then plain `wormhole send` will find it on your `$PATH`.

You probably *don't* want to use `sudo` when you run `pip`. This tends to create conflicts with the system python libraries.

On OS X, you may need to pre-install `pip`, and run `$ xcode-select --install` to get GCC, which is needed to compile the `libsodium` cryptography library during the installation process.

On Debian/Ubuntu systems, you may need to install some support libraries first:

`$ sudo apt-get install python-pip build-essential python-dev libffi-dev libssl-dev`

On Linux, if you get errors like `fatal error: sodium.h: No such file or directory`, either use `SODIUM_INSTALL=bundled pip install magic-wormhole`, or try installing the `libsodium-dev` / `libsodium-devel` package. These work around a bug in pynacl which gets confused when the libsodium runtime is installed (e.g. `libsodium13`) but not the development package.

On Windows, python2 may work better than python3. On older systems, `$ pip install --upgrade pip` may be necessary to get a version that can compile all the dependencies. Most of the dependencies are published as binary wheels, but in case your system is unable to find these, it will have to compile them, for which Microsoft Visual C++ 9.0 may be required.

1.3 Motivation

- Moving a file to a friend's machine, when the humans can speak to each other (directly) but the computers cannot
- Delivering a properly-random password to a new user via the phone
- Supplying an SSH public key for future login use

Copying files onto a USB stick requires physical proximity, and is uncomfortable for transferring long-term secrets because flash memory is hard to erase. Copying files with ssh/scp is fine, but requires previous arrangements and an account on the target machine, and how do you bootstrap the account? Copying files through email first requires transcribing an email address in the opposite direction, and is even worse for secrets, because email is unencrypted. Copying files through encrypted email requires bootstrapping a GPG key as well as an email address. Copying files through Dropbox is not secure against the Dropbox server and results in a large URL that must be transcribed. Using a URL shortener adds an extra step, reveals the full URL to the shortening service, and leaves a short URL that can be guessed by outsiders.

Many common use cases start with a human-mediated communication channel, such as IRC, IM, email, a phone call, or a face-to-face conversation. Some of these are basically secret, or are "secret enough" to last until the code is delivered and used. If this does not feel strong enough, users can turn on additional verification that doesn't depend upon the secrecy of the channel.

The notion of a "magic wormhole" comes from the image of two distant wizards speaking the same enchanted phrase at the same time, and causing a mystical connection to pop into existence between them. The wizards then throw books into the wormhole and they fall out the other side. Transferring files securely should be that easy.

1.4 Design

The `wormhole` tool uses PAKE "Password-Authenticated Key Exchange", a family of cryptographic algorithms that uses a short low-entropy password to establish a strong high-entropy shared key. This key can then be used to encrypt data. `wormhole` uses the SPAKE2 algorithm, due to Abdalla and Pointcheval[1].

PAKE effectively trades off interaction against offline attacks. The only way for a network attacker to learn the shared key is to perform a man-in-the-middle attack during the initial connection attempt, and to correctly guess the code being used by both sides. Their chance of doing this is inversely proportional to the entropy of the wormhole code. The default is to use a 16-bit code (use –code-length= to change this), so for each use of the tool, an attacker gets a 1-in-65536 chance of success. As such, users can expect to see many error messages before the attacker has a reasonable chance of success.

1.5 Timing

The program does not have any built-in timeouts, however it is expected that both clients will be run within an hour or so of each other. This makes the tool most useful for people who are having a real-time conversation already, and want to graduate to a secure connection. Both clients must be left running until the transfer has finished.

1.6 Relays

The wormhole library requires a "Mailbox Server" (also known as the "Rendezvous Server"): a simple WebSocket-based relay that delivers messages from one client to another. This allows the wormhole codes to omit IP addresses and port numbers. The URL of a public server is baked into the library for use as a default, and will be freely available until volume or abuse makes it infeasible to support. Applications which desire more reliability can easily run their own relay and configure their clients to use it instead. Code for the Mailbox Server is in a separate package named `magic-wormhole-mailbox-server` and has documentation here. Both clients must use the same mailbox server. The default can be overridden with the `--relay-url` option.

The file-transfer commands also use a "Transit Relay", which is another simple server that glues together two inbound TCP connections and transfers data on each to the other (the moral equivalent of a TURN server). The `wormhole send` file mode shares the IP addresses of each client with the other (inside the encrypted message), and both clients first attempt to connect directly. If this fails, they fall back to using the transit relay. As before, the host/port of a public server is baked into the library, and should be sufficient to handle moderate traffic. Code for the Transit Relay is provided a separate package named `magic-wormhole-transit-relay` with instructions here. The clients exchange transit relay information during connection negotiation, so they can be configured to use different ones without problems. Use the `--transit-helper` option to override the default.

The protocol includes provisions to deliver notices and error messages to clients: if either relay must be shut down, these channels will be used to provide information about alternatives.

1.7 CLI tool

- `wormhole send [args] --text TEXT`
- `wormhole send [args] FILENAME`
- `wormhole send [args] DIRNAME`
- `wormhole receive [args]`

Both commands accept additional arguments to influence their behavior:

- `--code-length WORDS`: use more or fewer than 2 words for the code
- `--verify` : print (and ask user to compare) extra verification string

1.7.1 Tab-Completion

Wormhole codes will tab-complete for receivers out-of-the-box.

If you desire shell tab-completion on sub-commands, we include generated files from Click for Bash, Zsh and Fish shells in wormhole_completion.bash (or `.zsh`, `.fish`). Put this file in your favourite location and add a line like `source ~/wormhole_completion.bash` to `~/.bashrc` (or similar for `zsh` and `fish` shells).

1.8 Library

The `wormhole` module makes it possible for other applications to use these code-protected channels. This includes Twisted support, and (in the future) will include blocking/synchronous support too. See docs/api.md for details.

The file-transfer tools use a second module named `wormhole.transit`, which provides an encrypted record-pipe. It knows how to use the Transit Relay as well as direct connections, and attempts them all in parallel. `TransitSender` and `TransitReceiver` are distinct, although once the connection is established, data can flow in either direction. All data is encrypted (using nacl/libsodium "secretbox") using a key derived from the PAKE phase. See `src/wormhole/cli/cmd_send.py` for examples.

1.9 Development

- Bugs and patches at the GitHub project page.
- Chat via IRC: #magic-wormhole on irc.libera.chat
- Chat via Matrix: #magic-wormhole on matrix.org

To set up Magic Wormhole for development, you will first need to install virtualenv.

Once you've done that, `git clone` the repo, `cd` into the root of the repository, and run:

```
virtualenv venv
source venv/bin/activate
pip install --upgrade pip setuptools
```

Now your virtualenv has been activated. You'll want to re-run `source venv/bin/activate` for every new terminal session you open.

To install Magic Wormhole and its development dependencies into your virtualenv, run:

```
pip install -e .[dev]
```

If you are using zsh, such as on macOS Catalina or later, you will have to run `pip install -e .'[dev]'` instead.

While the virtualenv is active, running `wormhole` will get you the development version.

1.9.1 Running Tests

Within your virtualenv, the command-line program `trial` will run the test suite:

```
trial wormhole
```

This tests the entire `wormhole` package. If you want to run only the tests for a specific module, or even just a specific test, you can specify it instead via Python's standard dotted import notation, e.g.:

```
trial wormhole.test.test_cli.PregeneratedCode.test_file_tor
```

Developers can also just clone the source tree and run `tox` to run the unit tests on all supported (and installed) versions of python: 2.7, 3.7 and 3.8.

1.9.2 Troubleshooting

Every so often, you might get a traceback with the following kind of error:

```
pkg_resources.DistributionNotFound: The 'magic-wormhole==0.9.1-268.g66e0d86.dirty'␣
↪distribution was not found and is required by the application
```

If this happens, run `pip install -e .[dev]` again.

1.9.3 Other

Relevant xkcd :-)

1.10 License, Compatibility

This library is released under the MIT license, see LICENSE for details.

This library is compatible with python2.7, 3.7 and 3.8 .

CHAPTER 2

The Magic-Wormhole Ecosystem

This page attempts to summarize the current state of various codebases that implement or extend the Magic Wormhole protocols. If you feel something is missing, please open an Issue or (better yet) a Pull Request at https://github.com/magic-wormhole/magic-wormhole

This document represents our best knowledge as of September, 2023.

2.1 Documentation

There are many documents in this repository itself, which tend to be more Python-specific. The Magic Wormhole Protocols repository aims to collect programming-language-agnostic documentation and specifications of the core Magic Wormhole protocol and extensions. Currently, it is still not complete, and sometimes fails to describe existing features or describes enhancements that don't yet have an implementation.

Rendered versions of this document exist at magic-wormhole.readthedocs.io.

2.2 Protocols Overview

There are several main pieces of the protocol:

- the core "mailbox protocol", spoken via the mailbox server (formerly called "rendezvous server" in some places) and client implementations
- the "transit relay" protocol, spoken by the transit relay server
- the Dilation protocol (which depends on the core mailbox protocol), spoken by client implementations

It is possible for clients to specify zero or many Transit Relays, but both peers must use the same Mailbox Server to successfully communicate (even if they end up with a direct, non-relay connection).

This document does not try to describe any of these protocols in detail.

2.3 Implementations and Support

There are several main features of the core protocol; while implementations strive for completeness, we document here what features those implementations actually support.

The only known Mailbox Server is https://github.com/magic-wormhole/magic-wormhole-mailbox-server The only known Transit Relay implementation is https://github.com/magic-wormhole/magic-wormhole-transit-relay

The Dilation protocol itself is still in development (with a basically-complete Python implementation, a servicable specification and an in-development Rust implementation).

The "Seeds" extension is a plan with no known implementations or specifications

The "Permissions" extension has a "proof-of-concept" Python client and server implementation and a specification but is not in any releases.

2.4 Features Supported by Implementations

The separate features represented in the below table are:

- *Core*: the basic client-visible features of the core mailbox protocol to *open*, *allocate*, *claim* and *close* a mailbox (as well as *add* messages to it)
- *Reconnect*: client state and behavior to re-connect to an in-progress or existing mailbox (e.g. in case of network failure, etc)
- *File Transfer v1* ("File v1"): the exsting file-transfer protocol (allowing single Files or Directories or text-messages to be transferred in one direction)
- *Permissions*: an extension to allow the server to request additional work or information from clients before allowing access.
- *Dilation*: the core Dilation protocol
- *Dilated File Transfer* ("Dilated Transfer"): a more fully-featured file-transfer protocol on top of *Dilation*.

Support is described as:

- *Full*: supports the feature
- *Partial*: supports the feature, with some caveats
- *Experimental*: supports the feature fully, but not quite final (e.g. needs flags to turn on, API may change, etc)
- *PoC*: some level of implementation exists, but not enough to be considered "Experimental".
- *No*: the feature is not supported

Table 1: Implementation Support

Language	Core	Reconnect	File v1	Dilation	Dilated Transfer
Python	Full	Full	Full	Experimental	PoC
Rust	Full	Partial	Partial	PoC	No
Haskell	Full	No	Full	No	No
Go	Full	???	Full	No	No

Notes: * the Rust implementation v1 file-transfer doesn't support text-messages, or directory transfer (although it will produce a tarball and send it, that is not automatically unpacked on the other side) * there are two parts to the Haskell implementation: a library, and a Haskell file-transfer CLI client

2.5 End User / Client Applications

Based on the above libraries, there are several end-user applications targeting different platforms. Unless otherwise noted, these "inherit" any limitations of their langauge's library implementation from the above table.

2.5.1 Library and CLI

- magic-wormhole the Python reference implementation and CLI (the command-line progam is called `wormhole` in most distributions)
- wormhole-william is a Go library and CLI for file-transfer
- magic-wormhole.rs provides a library and CLI for file-transfer
- haskell-magic-wormhole and wormhole-client are a library and CLI for file-transfer in Haskell
- dart bindings allowing Wormhole William to be used in Flutter.

2.5.2 GUIs for Desktop, Mobile, Web

- Warp is a GNOME GUI written in Rust
- Winden is a Web client and deployment (using the Go implemtation via WASM)
- Destiny is an Android (and iOS) app using Flutter (with the Go implementation for wormhole). Also on proprietary app stores.
- Wormhole for Android. Based on the Rust implementation.
- Mobile Wormhole for Android (also on f-droid. Based on the Python implementation, using Kivy
- Wormhole William Mobile for Android and iOS.
- Rymdport is a cross-platform graphical desktop application based on wormhole-william.

2.6 Integrations

These use the basic file-transfer functionality of the protocol, but build it in to some other application.

- tmux-wormhole a tmux plugin allowing use of file-transfer from within a tmux session (based on the Go implementation).
- termshark integrates `wormhole-william` (the Go implementation) to facilitate transfer of `.pcap` files (see the termshark User Guide

2.6.1 Other Uses

Some other interesting uses of Magic Wormhole that don't directly use the file-transfer protocol. If you know of others, please send them along!

- Port-forwarding: over the classic Transit protocol in the rust implementation and over the Dilation protocol in Python as fow (foward-over-wormhole).
- Invite / key-exchange: Magic Folder implements a custom protocol to do "introduction" / key-exchange.
- Invite / configuration exchange: Tahoe-LAFS uses Magic Wormhole to exchange configuration (and keys) for participants to join a Grid.

CHAPTER 3

Tor Support in Magic-Wormhole

The `wormhole` command-line tool has built-in support for performing transfers over Tor. To use it, you must install with the "tor" extra, like this:

```
pip install magic-wormhole[tor]
```

3.1 Usage

Just add `--tor` to use a running Tor daemon:

```
wormhole send --tor myfile.jpg

wormhole receive --tor
```

You should use `--tor` rather than running `wormhole` under tsocks or torsocks because the magic-wormhole "Transit" protocol normally sends the IP addresses of each computer to its peer, to attempt a direct connection between the two (somewhat like the FTP protocol would do). External tor-ifying programs don't know about this, so they can't strip these addresses out. Using `--tor` puts magic-wormhole into a mode where it does not share any IP addresses.

`--tor` causes the program to look for a Tor control port in the three most common locations:

- `unix:/var/run/tor/control`: Debian/Ubuntu Tor listen here
- `tcp:localhost:9051`: the standard Tor control port
- `tcp:localhost:9151`: control port for TorBrowser's embedded Tor

If `wormhole` is unable to establish a control-port connection to any of those locations, it will assume there is a SOCKS daemon listening on `tcp:localhost:9050`, and hope for the best (if no SOCKS daemon is available on that port, the initial Mailbox connection will fail, and the program will exit with an error before doing anything else).

The default behavior will Just Work if:

- you are on a Debian-like system, and the `tor` package is installed, or:
- you have launched the `tor` daemon manually, or:

- the TorBrowser application is running when you start `wormhole`

On Debian-like systems, if your account is a member of the `debian-tor` group, `wormhole` will use the control-port to ask for the right SOCKS port. If not, it should fall back to using the default SOCKS port on 9050. To add your account to the `debian-tor` group, use e.g. `sudo adduser MYUSER debian-tor`. Access to the control-port will be more significant in the future, when `wormhole` can listen on "onion services": see below for details.

3.2 Other Ways To Reach Tor

If `tor` is installed, but you cannot use the control-port or SOCKS-port for some reason, then you can use `--launch-tor` to ask `wormhole` to start a new Tor daemon for the duration of the transfer (and then shut it down afterwards). This will add 30-40 seconds to program startup.

```
wormhole send --tor --launch-tor myfile.jpg
```

Alternatively, if you know of a pre-existing Tor daemon with a non-standard control-port, you can specify that control port with the `--tor-control-port=` argument:

```
wormhole send --tor --tor-control-port=tcp:127.0.0.1:9251 myfile.jpg
```

3.3 .onion servers

In the future, `wormhole` with `--tor` will listen on an ephemeral "onion service" when file transfers are requested. If both sides are Tor-capable, this will allow transfers to take place "directly" (via the Tor network) from sender to receiver, bypassing the Transit Relay server. This will require access to a Tor control-port (to ask Tor to create a new ephemeral onion service). SOCKS-port access will not be sufficient.

However the current version of `wormhole` does not use onion services. For now, if both sides use `--tor`, any file transfers must use the transit relay, since neither side will advertise any listening IP addresses.

CHAPTER 4

Protocol/API/Library Introduction

The magic-wormhole (Python) distribution provides several things: an executable tool (”bin/wormhole”), an importable library (`import wormhole`), the URL of a publicly-available Mailbox Server, and the definition of a protocol used by all three.

The executable tool provides basic sending and receiving of files, directories, and short text strings. These all use `wormhole send` and `wormhole receive` (which can be abbreviated as `wormhole tx` and `wormhole rx`). It also has a mode to facilitate the transfer of SSH keys. This tool, while useful on its own, is just one possible use of the protocol.

The `wormhole` library provides an API to establish a bidirectional ordered encrypted record pipe to another instance (where each record is an arbitrary-sized bytestring). This does not provide file-transfer directly: the “bin/wormhole” tool speaks a simple protocol through this record pipe to negotiate and perform the file transfer.

`wormhole/cli/public_relay.py` contains the URLs of a Mailbox Server and a Transit Relay which I provide to support the file-transfer tools, which other developers should feel free to use for their applications as well. I cannot make any guarantees about performance or uptime for these servers: if you want to use Magic Wormhole in a production environment, please consider running a server on your own infrastructure (just run `wormhole-server start` and modify the URLs in your application to point at it).

4.1 The Magic-Wormhole Protocol

There are several layers to the protocol.

At the bottom level, each client opens a WebSocket to the Mailbox Server, sending JSON-based commands to the server, and receiving similarly-encoded messages. Some of these commands are addressed to the server itself, while others are instructions to queue a message to other clients, or are indications of messages coming from other clients. All these messages are described in “server-protocol.md”.

These inter-client messages are used to convey the PAKE protocol exchange, then a “VERSION” message (which doubles to verify the session key), then some number of encrypted application-level data messages. “client-protocol.md” describes these wormhole-to-wormhole messages.

Each wormhole-using application is then free to interpret the data messages as it pleases. The file-transfer app sends an “offer” from the `wormhole send` side, to which the `wormhole receive` side sends a response, after which the Transit connection is negotiated (if necessary), and finally the data is sent through the Transit connection. "file-transfer-protocol.md" describes this application's use of the client messages.

4.2 The `wormhole` API

Application use the `wormhole` library to establish wormhole connections and exchange data through them. Please see `api.md` for a complete description of this interface.

CHAPTER 5

The Magic-Wormhole API

This library provides a mechanism to securely transfer small amounts of data between two computers. Both machines must be connected to the internet, but they do not need to have public IP addresses or know how to contact each other ahead of time.

Security and connectivity is provided by means of a "wormhole code": a short string that is transcribed from one machine to the other by the users at the keyboard. This works in conjunction with a baked-in "mailbox server" that relays information from one machine to the other.

The "Wormhole" object provides a secure record pipe between any two programs that use the same wormhole code (and are configured with the same application ID and mailbox server). Each side can send multiple messages to the other, but the encrypted data for all messages must pass through (and be temporarily stored on) the mailbox server, which is a shared resource. For this reason, larger data (including bulk file transfers) should use the Transit class instead. The Wormhole can be used to create a Transit object for this purpose. In the future, Transit will be deprecated, and this functionality will be incorporated directly as a "dilated wormhole".

A quick example:

```python
import wormhole

async def example_initiator(reactor):
    w = wormhole.create(appid, relay_url, reactor)
    w.allocate_code()

    code = await w.get_code()
    print(f"code: {code}")

    w.send_message(b"outbound data")
    inbound = await w.get_message()
    await w.close()
```

5.1 Modes

The API comes in two flavors: Delegated and Deferred. Controlling the Wormhole and sending data is identical in both, but they differ in how inbound data and events are delivered to the application.

In Delegated mode, the Wormhole is given a "delegate" object, on which certain methods will be called when information is available (e.g. when the code is established, or when data messages are received). In Deferred mode, the Wormhole object has methods which return Deferreds that will fire at these same times.

Delegated mode:

```
class MyDelegate:
    def wormhole_got_code(self, code):
        print("code: %s" % code)
    def wormhole_got_message(self, msg): # called for each message
        print("got data, %d bytes" % len(msg))

w = wormhole.create(appid, relay_url, reactor, delegate=MyDelegate())
w.allocate_code()
```

Deferred mode:

```
async def example_initiator(reactor):
    appid = "lothar.com/example"
    relay_url = public_relay.MAILBOX_RELAY
    w = wormhole.create(appid, relay_url, reactor)
    w.allocate_code()

    code = await w.get_code()
    print(f"code: {code}")

    msg = await w.get_message() # gets exactly one message
    print(f"got msg: {len(msg)} bytes")
    result = await w.close()
    print(f"closed: {result}")
```

5.2 Application Identifier

Applications using this library must provide an "application identifier", a simple string that distinguishes one application from another. To ensure uniqueness, use a domain name. To use multiple apps for a single domain, append a URL-like slash and path, like `example.com/app1`. This string must be the same on both clients, otherwise they will not see each other. The invitation codes are scoped to the app-id. Note that the app-id must be unicode, not bytes.

Distinct app-ids reduce the size of the connection-id numbers. If fewer than ten Wormholes are active for a given app-id, the connection-id will only need to contain a single digit, even if some other app-id is currently using thousands of concurrent sessions.

5.3 Mailbox Servers

The library depends upon a "mailbox server" which is a service (on a public IP address) that delivers small encrypted messages from one client to the other. This must be the same for both clients, and is generally baked-in to the application source code or default config.

This library includes the URL of a public mailbox server run by the author. Application developers can use this one, or they can run their own (see the warner/magic-wormhole-mailbox-server repository) and configure their clients to use it instead. The URL of the public mailbox server is passed as a unicode string. Note that because the server actually speaks WebSockets, the URL starts with `ws:` (or `wss:`) instead of `http:`.

5.4 Wormhole Parameters

All wormholes must be created with at least three parameters:

- `appid`: a (unicode) string
- `relay_url`: a (unicode) string
- `reactor`: the Twisted reactor object

In addition to these three, the `wormhole.create()` function takes several optional arguments:

- `delegate`: provide a Delegate object to enable "delegated mode", or pass None (the default) to get "deferred mode"
- `journal`: provide a Journal object to enable journaled mode. See journal.md for details. Note that journals only work with delegated mode, not with deferred mode.
- `tor_manager`: to enable Tor support, create a `wormhole.TorManager` instance and pass it here. This will hide the client's IP address by proxying all connections (mailbox and transit) through Tor. It also enables connecting to Onion-service transit hints, and (in the future) will enable the creation of Onion-services for transit purposes.
- `timing`: this accepts a DebugTiming instance, mostly for internal diagnostic purposes, to record the transmit/receive timestamps for all messages. The `wormhole --dump-timing=` feature uses this to build a JSON-format data bundle, and the `misc/dump-timing.py` tool can build a scrollable timing diagram from these bundles.
- `welcome_handler`: this is a function that will be called when the Mailbox Server's "welcome" message is received. It is used to display important server messages in an application-specific way.
- `versions`: this can accept a dictionary (JSON-encodable) of data that will be made available to the peer via the `got_version` event. This data is delivered before any data messages, and can be used to indicate peer capabilities.

5.5 Code Management

Each wormhole connection is defined by a shared secret "wormhole code". These codes can be created by humans offline (by picking a unique number and some secret words), but are more commonly generated by asking the library to make one. In the "bin/wormhole" file-transfer tool, the default behavior is for the sender's program to create the code, and for the receiver to type it in.

The code is a (unicode) string in the form `NNN-code-words`. The numeric "NNN" prefix is the "channel id" or "nameplate", and is a short integer allocated by talking to the mailbox server. The rest is a randomly-generated selection from the PGP wordlist, providing a default of 16 bits of entropy. The initiating program should display this code to the user, who should transcribe it to the receiving user, who gives it to their local Wormhole object by calling `set_code()`. The receiving program can also use `input_code()` to use a readline-based input function: this offers tab completion of allocated channel-ids and known codewords.

The Wormhole object has three APIs for generating or accepting a code:

- `w.allocate_code(length=2)`: this contacts the Mailbox Server, allocates a short numeric nameplate, chooses a configurable number of random words, then assembles them into the code
- `w.set_code(code)`: this accepts the complete code as an argument
- `helper = w.input_code()`: this facilitates interactive entry of the code, with tab-completion. The helper object has methods to return a list of viable completions for whatever portion of the code has been entered so far. A convenience wrapper is provided to attach this to the `rlcompleter` function of libreadline.

No matter which mode is used, the `w.get_code()` Deferred (or `delegate.wormhole_got_code(code)` callback) will fire when the code is known. `get_code` is clearly necessary for `allocate_code`, since there's no other way to learn what code was created, but it may be useful in other modes for consistency.

The code-entry Helper object has the following API:

- `refresh_nameplates()`: requests an updated list of nameplates from the Mailbox Server. These form the first portion of the wormhole code (e.g. "4" in "4-purple-sausages"). Note that they are unicode strings (so "4", not 4). The Helper will get the response in the background, and calls to `get_nameplate_completions()` after the response will use the new list. Calling this after `h.choose_nameplate` will raise `AlreadyChoseNameplateError`.
- `matches = h.get_nameplate_completions(prefix)`: returns (synchronously) a set of completions for the given nameplate prefix, along with the hyphen that always follows the nameplate (and separates the nameplate from the rest of the code). For example, if the server reports nameplates 1, 12, 13, 24, and 170 are in use, `get_nameplate_completions("1")` will return `{"1-", "12-", "13-", "170-"}`. You may want to sort these before displaying them to the user. Raises `AlreadyChoseNameplateError` if called after `h.choose_nameplate`.
- `h.choose_nameplate(nameplate)`: accepts a string with the chosen nameplate. May only be called once, after which `AlreadyChoseNameplateError` is raised. (in this future, this might return a Deferred that fires (with None) when the nameplate's wordlist is known (which happens after the nameplate is claimed, requiring a roundtrip to the server)).
- `d = h.when_wordlist_is_available()`: return a Deferred that fires (with None) when the wordlist is known. This can be used to block a readline frontend which has just called `h.choose_nameplate()` until the resulting wordlist is known, which can improve the tab-completion behavior.
- `matches = h.get_word_completions(prefix)`: return (synchronously) a set of completions for the given words prefix. This will include a trailing hyphen if more words are expected. The possible completions depend upon the wordlist in use for the previously-claimed nameplate, so calling this before `choose_nameplate` will raise `MustChooseNameplateFirstError`. Calling this after `h.choose_words()` will raise `AlreadyChoseWordsError`. Given a prefix like "su", this returns a set of strings which are potential matches (e.g. `{"supportive-", "surrender-", "suspicious-"}`. The prefix should not include the nameplate, but *should* include whatever words and hyphens have been typed so far (the default wordlist uses alternate lists, where even numbered words have three syllables, and odd numbered words have two, so the completions depend upon how many words are present, not just the partial last word). E.g. `get_word_completions("pr")` will return `{"processor-", "provincial-", "proximate-"}`, while `get_word_completions("opulent-pr")` will return `{"opulent-preclude", "opulent-prefer", "opulent-preshrunk", "opulent-printer", "opulent-prowler"}` (note the lack of a trailing hyphen, because the wordlist is expecting a code of length two). If the wordlist is not yet known, this returns an empty set. All return values will `.startswith(prefix)`. The frontend is responsible for sorting the results before display.
- `h.choose_words(words)`: call this when the user is finished typing in the code. It does not return anything, but will cause the Wormhole's `w.get_code()` (or corresponding delegate) to fire, and triggers the wormhole connection process. This accepts a string like "purple-sausages", without the nameplate. It must be called after `h.choose_nameplate()` or `MustChooseNameplateFirstError` will be raised. May only be called once, after which `AlreadyChoseWordsError` is raised.

The `input_with_completion` wrapper is a function that knows how to use the code-entry helper to do tab completion of wormhole codes:

```
from wormhole import create, input_with_completion

async def example(reactor):
    w = create(appid, relay_url, reactor)
    input_with_completion("Wormhole code:", w.input_code(), reactor)
    code = await w.get_code()
```

This helper runs python's (raw) `input()` function inside a thread, since `input()` normally blocks.

The two machines participating in the wormhole setup are not distinguished: it doesn't matter which one goes first, and both use the same Wormhole constructor function. However if `w.allocate_code()` is used, only one side should use it.

Providing an invalid nameplate (which is easily caused by cut-and-paste errors that include an extra space at the beginning, or which copy the words but not the number) will raise a `KeyFormatError`, either in `w.set_code(code)` or in `h.choose_nameplate()`.

5.6 Offline Codes

In most situations, the "sending" or "initiating" side will call `w.allocate_code()` and display the resulting code. The sending human reads it and speaks, types, performs charades, or otherwise transmits the code to the receiving human. The receiving human then types it into the receiving computer, where it either calls `w.set_code()` (if the code is passed in via argv) or `w.input_code()` (for interactive entry).

Usually one machine generates the code, and a pair of humans transcribes it to the second machine (so `w.allocate_code()` on one side, and `w.set_code()` or `w.input_code()` on the other). But it is also possible for the humans to generate the code offline, perhaps at a face-to-face meeting, and then take the code back to their computers. In this case, `w.set_code()` will be used on both sides. It is unlikely that the humans will restrict themselves to a pre-established wordlist when manually generating codes, so the completion feature of `w.input_code()` is not helpful.

When the humans create an invitation code out-of-band, they are responsible for choosing an unused channel-ID (simply picking a random 3-or-more digit number is probably enough), and some random words. Dice, coin flips, shuffled cards, or repeated sampling of a high-resolution stopwatch are all useful techniques. The invitation code uses the same format either way: channel-ID, a hyphen, and an arbitrary string. There is no need to encode the sampled random values (e.g. by using the Diceware wordlist) unless that makes it easier to transcribe: e.g. rolling 6 dice could result in a code like "913-166532", and flipping 16 coins could result in "123-HTTHHHTTHTTHHTHH".

5.7 Welcome Messages

The first message sent by the mailbox server is a "welcome" message (a dictionary). This is sent as soon as the client connects to the server, generally before the code is established. Clients should use `await get_welcome()` to get and process the `motd` key (and maybe `current_cli_version`) inside the welcome message.

The welcome message serves three main purposes:

- notify users about important server changes, such as CAPTCHA requirements driven by overload, or donation requests
- enable future protocol negotiation between clients and the server
- advise users of the CLI tools (`wormhole send`) to upgrade to a new version

There are three keys currently defined for the welcome message, all of which are optional (the welcome message omits “error” and “motd” unless the server operator needs to signal a problem).

- `motd`: if this key is present, it will be a string with embedded newlines. The client should display this string to the user, including a note that it comes from the magic-wormhole Mailbox Server and that server’s URL.
- `error`: if present, the server has decided it cannot service this client. The string will be wrapped in a `WelcomeError` (which is a subclass of `WormholeError`), and all API calls will signal errors (pending Deferreds will errback). The mailbox connection will be closed.
- `current_cli_version`: if present, the server is advising instances of the CLI tools (the `wormhole` command included in the python distribution) that there is a newer release available, thus users should upgrade if they can, because more features will be available if both clients are running the same version. The CLI tools compare this string against their `__version__` and can print a short message to stderr if an upgrade is warranted.

The main idea of `error` is to allow the server to cleanly inform the client about some necessary action it didn’t take. The server currently sends the welcome message as soon as the client connects (even before it receives the “claim” request), but a future server could wait for a required client message and signal an error (via the Welcome message) if it didn’t see this extra message before the CLAIM arrived.

This could enable changes to the protocol, e.g. requiring a CAPTCHA or proof-of-work token when the server is under DoS attack. The new server would send the current requirements in an initial message (which old clients would ignore). New clients would be required to send the token before their “claim” message. If the server sees “claim” before “token”, it knows that the client is too old to know about this protocol, and it could send a “welcome” with an `error` field containing instructions (explaining to the user that the server is under attack, and they must either upgrade to a client that can speak the new protocol, or wait until the attack has passed). Either case is better than an opaque exception later when the required message fails to arrive.

(Note that the server can also send an explicit ERROR message at any time, and the client should react with a ServerError. Versions 0.9.2 and earlier of the library did not pay attention to the ERROR message, hence the server should deliver errors in a WELCOME message if at all possible)

The `error` field is handled internally by the Wormhole object. The other fields can be processed by application, by using `d=w.get_welcome()`. The Deferred will fire with the full welcome dictionary, so any other keys that a future server might send will be available to it.

Applications which need to display `motd` or an upgrade message, and wish to do so before using stdin/stdout for interactive code entry (`w.input_code()`) should wait for `get_welcome()` before starting the entry process:

```python
async def go():
    w = wormhole.create(appid, relay_url, reactor)
    welcome = await w.get_welcome()
    if "motd" in welcome:
        print(welcome["motd"])
    input_with_completion("Wormhole code:", w.input_code(), reactor)
    ...
```

5.8 Verifier

For extra protection against guessing attacks, Wormhole can provide a “Verifier”. This is a moderate-length series of bytes (a SHA256 hash) that is derived from the supposedly-shared session key. If desired, both sides can display this value, and the humans can manually compare them before allowing the rest of the protocol to proceed. If they do not match, then the two programs are not talking to each other (they may both be talking to a man-in-the-middle attacker), and the protocol should be abandoned.

Deferred-mode applications can wait for `d=w.get_verifier()`: the Deferred it returns will fire with the verifier. You can turn this into hex or Base64 to print it, or render it as ASCII-art, etc.

Asking the wormhole object for the verifier does not affect the flow of the protocol. To benefit from verification, applications must refrain from sending any data (with `w.send_message(data)`) until after the verifiers are approved by the user. In addition, applications must queue or otherwise ignore incoming (received) messages until that point. However once the verifiers are confirmed, previously-received messages can be considered valid and processed as usual.

5.9 Events

As the wormhole connection is established, several events may be dispatched to the application. In Delegated mode, these are dispatched by calling functions on the delegate object. In Deferred mode, the application retrieves Deferred objects from the wormhole, and event dispatch is performed by firing those Deferreds.

Most applications will only use `code`, `received`, and `close`.

- code (`code = yield w.get_code()` / `dg.wormhole_got_code(code)`): fired when the wormhole code is established, either after `w.allocate_code()` finishes the generation process, or when the Input Helper returned by `w.input_code()` has been told `h.set_words()`, or immediately after `w.set_code(code)` is called. This is most useful after calling `w.allocate_code()`, to show the generated code to the user so they can transcribe it to their peer.
- key (`yield w.get_unverified_key()` / `dg.wormhole_got_unverified_key(key)`): fired (with the raw master SPAKE2 key) when the key-exchange process has completed and a purported shared key is established. At this point we do not know that anyone else actually shares this key: the peer may have used the wrong code, or may have disappeared altogether. To wait for proof that the key is shared, wait for `get_verifier` instead. This event is really only useful for detecting that the initiating peer has disconnected after leaving the initial PAKE message, to display a pacifying message to the user.
- verifier (`verifier = yield w.get_verifier()` / `dg.wormhole_got_verifier(verifier)`: fired when the key-exchange process has completed and a valid VERSION message has arrived. The "verifier" is a byte string with a hash of the shared session key; clients can compare them (probably as hex) to ensure that they're really talking to each other, and not to a man-in-the-middle. When `get_verifier` happens, this side knows that *someone* has used the correct wormhole code; if someone used the wrong code, the VERSION message cannot be decrypted, and the wormhole will be closed instead.
- versions (`versions = yield w.get_versions()` / `dg.wormhole_got_versions(versions)`: fired when the VERSION message arrives from the peer. This fires just after `verified`, but delivers the "app_versions" data (as passed into `wormhole.create(versions=)`) instead of the verifier string. This is mostly a hack to make room for forwards-compatible changes to the CLI file-transfer protocol, which sends a request in the first message (rather than merely sending the abilities of each side).
- received (`yield w.get_message()` / `dg.wormhole_got_message(msg)`: fired each time a data message arrives from the peer, with the bytestring that the peer passed into `w.send_message(msg)`. This is the primary data-transfer API.
- closed (`yield w.close()` / `dg.wormhole_closed(result)`: fired when `w.close()` has finished shutting down the wormhole, which means all nameplates and mailboxes have been deallocated, and the WebSocket connection has been closed. This also fires if an internal error occurs (specifically WrongPasswordError, which indicates that an invalid encrypted message was received), which also shuts everything down. The `result` value is an exception (or Failure) object if the wormhole closed badly, or a string like "happy" if it had no problems before shutdown.

5.10 Sending Data

The main purpose of a Wormhole is to send data. At any point after construction, callers can invoke `w.send_message(msg)`. This will queue the message if necessary, but (if all goes well) will eventually result in the peer getting a `received` event and the data being delivered to the application.

Since Wormhole provides an ordered record pipe, each call to `w.send_message` will result in exactly one `received` event on the far side. Records are not split, merged, dropped, or reordered.

Each side can do an arbitrary number of `send_message()` calls. The Wormhole is not meant as a long-term communication channel, but some protocols work better if they can exchange an initial pair of messages (perhaps offering some set of negotiable capabilities), and then follow up with a second pair (to reveal the results of the negotiation). The Mailbox Server does not currently enforce any particular limits on number of messages, size of messages, or rate of transmission, but in general clients are expected to send fewer than a dozen messages, of no more than perhaps 20kB in size (remember that all these messages are temporarily stored in a SQLite database on the server). A future version of the protocol may make these limits more explicit, and will allow clients to ask for greater capacity when they connect (probably by passing additional "mailbox attribute" parameters with the `allocate`/`claim`/`open` messages).

For bulk data transfer, see "transit.md", or the "Dilation" section below.

5.11 Closing

When the application is done with the wormhole, it should call `w.close()`, and wait for a `closed` event. This ensures that all server-side resources are released (allowing the nameplate to be re-used by some other client), and all network sockets are shut down.

In Deferred mode, this just means waiting for the Deferred returned by `w.close()` to fire. In Delegated mode, this means calling `w.close()` (which doesn't return anything) and waiting for the delegate's `wormhole_closed()` method to be called.

`w.close()` will errback (with some form of `WormholeError`) if anything went wrong with the process, such as:

- `WelcomeError`: the server told us to signal an error, probably because the client is too old understand some new protocol feature
- `ServerError`: the server rejected something we did
- `LonelyError`: we didn't hear from the other side, so no key was established
- `WrongPasswordError`: we received at least one incorrectly-encrypted message. This probably indicates that the other side used a different wormhole code than we did, perhaps because of a typo, or maybe an attacker tried to guess your code and failed.

If the wormhole was happy at the time it was closed, the `w.close()` Deferred will callback (probably with the string "happy", but this may change in the future).

5.12 Serialization

(NOTE: this section is speculative: this code has not yet been written)

Wormhole objects can be serialized. This can be useful for apps which save their own state before shutdown, and restore it when they next start up again.

The `w.serialize()` method returns a dictionary which can be JSON encoded into a unicode string (most applications will probably want to UTF-8 -encode this into a bytestring before saving on disk somewhere).

To restore a Wormhole, call `wormhole.from_serialized(data, reactor, delegate)`. This will return a wormhole in roughly the same state as was serialized (of course all the network connections will be disconnected).

Serialization only works for delegated-mode wormholes (since Deferreds point at functions, which cannot be serialized easily). It also only works for "non-dilated" wormholes (see below).

To ensure correct behavior, serialization should probably only be done in "journaled mode". See journal.md for details.

If you use serialization, be careful to never use the same partial wormhole object twice.

5.13 Dilation

(NOTE: this API is still in development)

To send bulk data, or anything more than a handful of messages, a Wormhole can be "dilated" into a form that uses a direct TCP connection between the two endpoints.

All wormholes start out "undilated". In this state, all messages are queued on the Mailbox Server for the lifetime of the wormhole, and server-imposed number/size/rate limits apply. Calling `w.dilate()` initiates the dilation process, and eventually yields a set of Endpoints. Once dilated these endpoints can be used to establish multiple (encrypted) "subchannel" connections to the other side.

Each subchannel behaves like a regular Twisted `ITransport`, so they can be glued to the Protocol instance of your choice. They also implement the IConsumer/IProducer interfaces.

These subchannels are *durable*: as long as the processes on both sides keep running, the subchannel will survive the network connection being dropped. For example, a file transfer can be started from a laptop, then while it is running, the laptop can be closed, moved to a new wifi network, opened back up, and the transfer will resume from the new IP address.

What's good about a non-dilated wormhole?:

- setup is faster: no delay while it tries to make a direct connection
- works with "journaled mode", allowing progress to be made even when both sides are never online at the same time, by serializing the wormhole

What's good about dilated wormholes?:

- they support bulk data transfer
- you get flow control (backpressure), and IProducer/IConsumer
- throughput is faster: no store-and-forward step

Use non-dilated wormholes when your application only needs to exchange a couple of messages, for example to set up public keys or provision access tokens. Use a dilated wormhole to move files, stream data, etc

Dilated wormholes can provide multiple "subchannels": these are multiplexed through the single (encrypted) TCP connection. Each subchannel is a separate stream (offering IProducer/IConsumer for flow control), and is opened and closed independently. A special "control channel" is available to both sides so they can coordinate how they use the subchannels.

The `d = w.dilate()` Deferred fires with a triple of Endpoints:

```python
d = w.dilate()
def _dilated(res):
    (control_channel_ep, subchannel_client_ep, subchannel_server_ep) = res
d.addCallback(_dilated)
```

The `control_channel_ep` endpoint is a client-style endpoint, so both sides will connect to it with `ep.connect(factory)`. This endpoint is single-use: calling `.connect()` a second time will fail. The control channel is symmetric: it doesn't matter which side is the application-level client/server or initiator/responder, if the application even has such concepts. The two applications can use the control channel to negotiate who goes first, if necessary.

The subchannel endpoints are *not* symmetric: for each subchannel, one side must listen as a server, and the other must connect as a client. Subchannels can be established by either side at any time. This supports e.g. bidirectional file transfer, where either user of a GUI app can drop files into the "wormhole" whenever they like.

The `subchannel_client_ep` on one side is used to connect to the other side's `subchannel_server_ep`, and vice versa. The client endpoint is reusable. The server endpoint is single-use: `.listen(factory)` may only be called once.

Applications are under no obligation to use subchannels: for many use cases, the control channel is enough.

To use subchannels, once the wormhole is dilated and the endpoints are available, the listening-side application should attach a listener to the `subchannel_server_ep` endpoint:

```
def _dilated(res):
    (control_channel_ep, subchannel_client_ep, subchannel_server_ep) = res
    f = Factory(MyListeningProtocol)
    subchannel_server_ep.listen(f)
```

When the connecting-side application wants to connect to that listening protocol, it should use `.connect()` with a suitable connecting protocol factory:

```
def _connect():
    f = Factory(MyConnectingProtocol)
    subchannel_client_ep.connect(f)
```

For a bidirectional file-transfer application, both sides will establish a listening protocol. Later, if/when the user drops a file on the application window, that side will initiate a connection, use the resulting subchannel to transfer the single file, and then close the subchannel.

```
def FileSendingProtocol(internet.Protocol):
    def __init__(self, metadata, filename):
        self.file_metadata = metadata
        self.file_name = filename
    def connectionMade(self):
        self.transport.write(self.file_metadata)
        sender = protocols.basic.FileSender()
        f = open(self.file_name,"rb")
        d = sender.beginFileTransfer(f, self.transport)
        d.addBoth(self._done, f)
    def _done(res, f):
        self.transport.loseConnection()
        f.close()
def _send(metadata, filename):
    f = protocol.ClientCreator(reactor,
                               FileSendingProtocol, metadata, filename)
    subchannel_client_ep.connect(f)
def FileReceivingProtocol(internet.Protocol):
    state = INITIAL
    def dataReceived(self, data):
        if state == INITIAL:
            self.state = DATA
            metadata = parse(data)
```

(continues on next page)

(continued from previous page)

```
            self.f = open(metadata.filename, "wb")
        else:
            # local file writes are blocking, so don't bother with IConsumer
            self.f.write(data)
    def connectionLost(self, reason):
        self.f.close()
def _dilated(res):
    (control_channel_ep, subchannel_client_ep, subchannel_server_ep) = res
    f = Factory(FileReceivingProtocol)
    subchannel_server_ep.listen(f)
```

5.14 Bytes, Strings, Unicode, and Python 3

All cryptographically-sensitive parameters are passed as bytes (”str” in python2, “bytes” in python3):

- verifier string
- data in/out
- transit records in/out

Other (human-facing) values are always unicode (”unicode” in python2, “str” in python3):

- wormhole code
- relay URL
- transit URLs
- transit connection hints (e.g. “host:port”)
- application identifier
- derived-key “purpose” string: `w.derive_key(PURPOSE, LENGTH)`

5.15 Full API list

action | Deferred-Mode | Delegated-Mode ——————— | ——————— | ————— . | d=w.get_welcome() | dg.wormhole_got_welcome(welcome) w.allocate_code() | | h=w.input_code() | | w.set_code(code) | | . | d=w.get_code() | dg.wormhole_got_code(code) . | d=w.get_unverified_key() | dg.wormhole_got_unverified_key(key) . | d=w.get_verifier() | dg.wormhole_got_verifier(verifier) . | d=w.get_versions() | dg.wormhole_got_versions(versions) key=w.derive_key(purpose, length) | | w.send_message(msg) | | . | d=w.get_message() | dg.wormhole_got_message(msg) w.close() | | dg.wormhole_closed(result) . | d=w.close() |

CHAPTER 6

Transit Protocol

The Transit protocol is responsible for establishing an encrypted bidirectional record stream between two programs. It must be given a “transit key” and a set of “hints” which help locate the other end (which are both delivered by Wormhole).

The protocol tries hard to create a **direct** connection between the two ends, but if that fails, it uses a centralized relay server to ferry data between two separate TCP streams (one to each client). Direct connection hints are used for the first, and relay hints are used for the second.

The current implementation starts with the following:

- detect all of the host’s IP addresses
- listen on a random TCP port
- offers the (address,port) pairs as hints

The other side will attempt to connect to each of those ports, as well as listening on its own socket. After a few seconds without success, they will both connect to a relay server.

6.1 Roles

The Transit protocol has pre-defined “Sender” and “Receiver” roles (unlike Wormhole, which is symmetric/nobody-goes-first). Each connection must have exactly one Sender and exactly one Receiver.

The connection itself is bidirectional: either side can send or receive records. However the connection establishment mechanism needs to know who is in charge, and the encryption layer needs a way to produce separate keys for each side..

This may be relaxed in the future, much as Wormhole was.

6.2 Records

Transit establishes a **record-pipe**, so the two sides can send and receive whole records, rather than unframed bytes. This is a side-effect of the encryption (which uses the NaCl "secretbox" function). The encryption adds 44 bytes of overhead to each record (4-byte length, 24-byte nonce, 32-byte MAC), so you might want to use slightly larger records for efficiency. The maximum record size is 2^32 bytes (4GiB). The whole record must be held in memory at the same time, plus its ciphertext, so very large ciphertexts are not recommended.

Transit provides **confidentiality**, **integrity**, and **ordering** of records. Passive attackers can only do the following:

- learn the size and transmission time of each record
- learn the sending and destination IP addresses

In addition, an active attacker is able to:

- delay delivery of individual records, while maintaining ordering (if they delay record #4, they must delay #5 and later as well)
- terminate the connection at any time

If either side receives a corrupted or out-of-order record, they drop the connection. Attackers cannot modify the contents of a record, or change the order of the records, without being detected and the connection being dropped. If a record is lost (e.g. the receiver observes records #1,#2,#4, but not #3), the connection is dropped when the unexpected sequence number is received.

6.3 Handshake

The transit key is used to derive several secondary keys. Two of them are used as a "handshake", to distinguish correct Transit connections from other programs that happen to connect to the Transit sockets by mistake or malice.

The handshake is also responsible for choosing exactly one TCP connection to use, even though multiple outbound and inbound connections are being attempted.

The SENDER-HANDSHAKE is the string `transit sender %s ready\n\n`, with the `%s` replaced by a hex-encoded 32-byte HKDF derivative of the transit key, using a "context string" of `transit_sender`. The RECEIVER-HANDSHAKE is the same but with `receiver` instead of `sender` (both for the string and the HKDF context).

The handshake protocol is like this:

- immediately upon connection establishment, the Sender writes SENDER-HANDSHAKE to the socket (regardless of whether the Sender initiated the TCP connection, or was listening on a socket and accepted the connection)
- likewise the Receiver immediately writes RECEIVER-HANDSHAKE to either kind of socket
- if the Sender sees anything other than RECEIVER-HANDSHAKE as the first bytes on the wire, it hangs up
- likewise with the Receiver and SENDER-HANDSHAKE
- if the Sender sees that this is the first connection to get RECEIVER-HANDSHAKE, it sends `go\n`. If some other connection got there first, it hangs up (or sends `nevermind\n` and then hangs up, but this is mostly for debugging, and implementations should not depend upon it). After sending `go`, it switches to encrypted-record mode.
- if the Receiver sees `go\n`, it switches to encrypted-record mode. If the receiver sees anything else, or a disconnected socket, it disconnects.

To tolerate the inevitable race conditions created by multiple contending sockets, only the Sender gets to decide which one wins: the first one to make it past negotiation. Hopefully this is correlated with the fastest connection pathway. The protocol ignores any socket that is not somewhat affiliated with the matching Transit instance.

Hints will frequently point to local IP addresses (local to the other end) which might be in use by unrelated nearby computers. The handshake helps to ignore these spurious connections. It is still possible for an attacker to cause the connection to fail, by intercepting both connections (to learn the two handshakes), then making new connections to play back the recorded handshakes, but this level of attacker could simply drop the user's packets directly.

Any participant in a Transit connection (i.e. the party on the other end of your wormhole) can cause their peer to make a TCP connection (and send the handshake string) to any IP address and port of their choosing. The handshake protocol is intended to make this no more than a minor nuisance.

6.4 Relay

The **Transit Relay** is a host which offers TURN-like services for magic-wormhole instances. It uses a TCP-based protocol with a handshake to determine which connection wants to be connected to which.

When connecting to a relay, the Transit client first writes RELAY-HANDSHAKE to the socket, which is `please relay %s\n`, where `%s` is the hex-encoded 32-byte HKDF derivative of the transit key, using `transit_relay_token` as the context. The client then waits for `ok\n`.

The relay waits for a second connection that uses the same token. When this happens, the relay sends `ok\n` to both, then wires the connections together, so that everything received after the token on one is written out (after the ok) on the other. When either connection is lost, the other will be closed (the relay does not support "half-close").

When clients use a relay connection, they perform the usual sender/receiver handshake just after the `ok\n` is received: until that point they pretend the connection doesn't even exist.

Direct connections are better, since they are faster and less expensive for the relay operator. If there are any potentially-viable direct connection hints available, the Transit instance will wait a few seconds before attempting to use the relay. If it has no viable direct hints, it will start using the relay right away. This prefers direct connections, but doesn't introduce completely unnecessary stalls.

The Transit client can attempt connections to multiple relays, and uses the first one that passes negotiation. Each side combines a locally-configured hostname/port (usually "baked in" to the application, and hosted by the application author) with additional hostname/port pairs that come from the peer. This way either side can suggest the relays to use. The `wormhole` application accepts a `--transit-helper tcp:myrelay.example.org:12345` command-line option to supply an additional relay. The connection hints provided by the Transit instance include the locally-configured relay, along with the dynamically-determined direct hints. Both should be delivered to the peer.

6.5 API

The Transit API uses Twisted and returns Deferreds for any call that cannot be handled immediately. The complete example is here:

```python
from twisted.internet.defer import inlineCallbacks
from wormhole.transit import TransitSender

@inlineCallbacks
def do_transit():
    s = TransitSender("tcp:relayhost.example.org:12345")
    my_connection_hints = yield s.get_connection_hints()
    # (send my hints via wormhole)
```

(continues on next page)

(continued from previous page)

```
    # (get their hints via wormhole)
    s.add_connection_hints(their_connection_hints)
    key = w.derive_key(application_id + "/transit-key")
    s.set_transit_key(key)
    rp = yield s.connect()
    rp.send_record(b"my first record")
    their_record = yield rp.receive_record()
    rp.send_record(b"Greatest Hits)
    other = yield rp.receive_record()
    yield rp.close()
```

First, create a Transit instance, giving it the connection information of the "baked-in" transit relay. The application must know whether it should use a Sender or a Receiver:

```
from wormhole.transit import TransitSender
s = TransitSender(baked_in_relay)
```

Next, ask the Transit for its direct and relay hints. This should be delivered to the other side via a Wormhole message (i.e. add them to a dict, serialize it with JSON, send the result as a message with `wormhole.send()`). The `get_connection_hints` method returns a Deferred, so in the example we use `@inlineCallbacks` to `yield` the result.

```
my_connection_hints = yield s.get_connection_hints()
```

Then, perform the Wormhole exchange, which ought to give you the direct and relay hints of the other side. Tell your Transit instance about their hints.

```
s.add_connection_hints(their_connection_hints)
```

Then use `wormhole.derive_key()` to obtain a shared key for Transit purposes, and tell your Transit about it. Both sides must use the same derivation string, and this string must not be used for any other purpose, but beyond that it doesn't much matter what the exact derivation string is. The key is secret, of course.

```
key = w.derive_key(application_id + "/transit-key")
s.set_transit_key(key)
```

Finally, tell the Transit instance to connect. This returns a Deferred that will yield a "record pipe" object, on which records can be sent and received. If no connection can be established within a timeout (defaults to 30 seconds), `connect()` will signal a Failure instead. The pipe can be closed with `close()`, which returns a Deferred that fires when all data has been flushed.

```
rp = yield s.connect()
rp.send_record(b"my first record")
their_record = yield rp.receive_record()
rp.send_record(b"Greatest Hits)
other = yield rp.receive_record()
yield rp.close()
```

Records can be sent and received in arbitrary order (you are not limited to taking turns).

The record-pipe object also implements the `IConsumer`/`IProducer` protocols for **bytes**, which means you can transfer a file by wiring up a file reader as a Producer. Each chunk of bytes that the Producer generates will be put into a single record. The Consumer interface works the same way. This enables backpressure and flow-control: if the far end (or the network) cannot keep up with the stream of data, the sender will wait for them to catch up before filling buffers without bound.

CHAPTER 7

Mailbox Server Protocol

7.1 Concepts

The Mailbox Server provides queued delivery of binary messages from one client to a second, and vice versa. Each message contains a "phase" (a string) and a body (bytestring). These messages are queued in a "Mailbox" until the other side connects and retrieves them, but are delivered immediately if both sides are connected to the server at the same time.

Mailboxes are identified by a large random string. "Nameplates", in contrast, have short numeric identities: in a wormhole code like "4-purple-sausages", the "4" is the nameplate.

Each client has a randomly-generated "side", a short hex string, used to differentiate between echoes of a client's own message, and real messages from the other client.

7.2 Application IDs

The server isolates each application from the others. Each client provides an "App Id" when it first connects (via the "BIND" message), and all subsequent commands are scoped to this application. This means that nameplates (described below) and mailboxes can be re-used between different apps. The AppID is a unicode string. Both sides of the wormhole must use the same AppID, of course, or they'll never see each other. The server keeps track of which applications are in use for maintenance purposes.

Each application should use a unique AppID. Developers are encouraged to use "DNSNAME/APPNAME" to obtain a unique one: e.g. the `bin/wormhole` file-transfer tool uses `lothar.com/wormhole/text-or-file-xfer`.

7.3 WebSocket Transport

At the lowest level, each client establishes (and maintains) a WebSocket connection to the Mailbox Server. If the connection is lost (which could happen because the server was rebooted for maintenance, or because the client's network connection migrated from one network to another, or because the resident network gremlins decided to mess with you today), clients should reconnect after waiting a random (and exponentially-growing) delay. The Python implementation waits about 1 second after the first connection loss, growing by 50% each time, capped at 1 minute.

Each message to the server is a dictionary, with at least a `type` key, and other keys that depend upon the particular message type. Messages from server to client follow the same format.

`misc/dump-timing.py` is a debug tool which renders timing data gathered from the server and both clients, to identify protocol slowdowns and guide optimization efforts. To support this, the client/server messages include additional keys. Client->Server messages include a random `id` key, which is copied into the `ack` that is immediately sent back to the client for all commands (logged for the timing tool but otherwise ignored). Some client->server messages (`list`, `allocate`, `claim`, `release`, `close`, `ping`) provoke a direct response by the server: for these, `id` is copied into the response. This helps the tool correlate the command and response. All server->client messages have a `server_tx` timestamp (seconds since epoch, as a float), which records when the message left the server. Direct responses include a `server_rx` timestamp, to record when the client's command was received. The tool combines these with local timestamps (recorded by the client and not shared with the server) to build a full picture of network delays and round-trip times.

All messages are serialized as JSON, encoded to UTF-8, and the resulting bytes sent as a single "binary-mode" WebSocket payload.

Servers can signal `error` for any message type it does not recognize. Clients and Servers must ignore unrecognized keys in otherwise-recognized messages. Clients must ignore unrecognized message types from the Server.

7.4 Connection-Specific (Client-to-Server) Messages

The first thing each client sends to the server, immediately after the WebSocket connection is established, is a `bind` message. This specifies the AppID and side (in keys `appid` and `side`, respectively) that all subsequent messages will be scoped to. While technically each message could be independent (with its own `appid` and `side`), I thought it would be less confusing to use exactly one WebSocket per logical wormhole connection.

The first thing the server sends to each client is the `welcome` message. This is intended to deliver important status information to the client that might influence its operation. The Python client currently reacts to the following keys (and ignores all others):

- `current_cli_version`: prompts the user to upgrade if the server's advertised version is greater than the client's version (as derived from the git tag)
- `motd`: prints this message, if present; intended to inform users about performance problems, scheduled downtime, or to beg for donations to keep the server running
- `error`: causes the client to print the message and then terminate. If a future version of the protocol requires a rate-limiting CAPTCHA ticket or other authorization record, the server can send `error` (explaining the requirement) if it does not see this ticket arrive before the `bind`.

A `ping` will provoke a `pong`: these are only used by unit tests for synchronization purposes (to detect when a batch of messages have been fully processed by the server). NAT-binding refresh messages are handled by the WebSocket layer (by asking Autobahn to send a keepalive messages every 60 seconds), and do not use `ping`.

If any client->server command is invalid (e.g. it lacks a necessary key, or was sent in the wrong order), an `error` response will be sent, This response will include the error string in the `error` key, and a full copy of the original message dictionary in `orig`.

7.5 Nameplates

Wormhole codes look like `4-purple-sausages`, consisting of a number followed by some random words. This number is called a "Nameplate".

On the Mailbox Server, the Nameplate contains a pointer to a Mailbox. Clients can "claim" a nameplate, and then later "release" it. Each claim is for a specific side (so one client claiming the same nameplate multiple times only counts as one claim). Nameplates are deleted once the last client has released it, or after some period of inactivity.

Clients can either make up nameplates themselves, or (more commonly) ask the server to allocate one for them. Allocating a nameplate automatically claims it (to avoid a race condition), but for simplicity, clients send a claim for all nameplates, even ones which they've allocated themselves.

Nameplates (on the server) must live until the second client has learned about the associated mailbox, after which point they can be reused by other clients. So if two clients connect quickly, but then maintain a long-lived wormhole connection, they do not need to consume the limited space of short nameplates for that whole time.

The `allocate` command allocates a nameplate (the server returns one that is as short as possible), and the `allocated` response provides the answer. Clients can also send a `list` command to get back a `nameplates` response with all allocated nameplates for the bound AppID: this helps the code-input tab-completion feature know which prefixes to offer. The `nameplates` response returns a list of dictionaries, one per claimed nameplate, with at least an `id` key in each one (with the nameplate string). Future versions may record additional attributes in the nameplate records, specifically a wordlist identifier and a code length (again to help with code-completion on the receiver).

7.6 Mailboxes

The server provides a single "Mailbox" to each pair of connecting Wormhole clients. This holds an unordered set of messages, delivered immediately to connected clients, and queued for delivery to clients which connect later. Messages from both clients are merged together: clients use the included `side` identifier to distinguish echoes of their own messages from those coming from the other client.

Each mailbox is "opened" by some number of clients at a time, until all clients have closed it. Mailboxes are kept alive by either an open client, or a Nameplate which points to the mailbox (so when a Nameplate is deleted from inactivity, the corresponding Mailbox will be too).

The `open` command both marks the mailbox as being opened by the bound side, and also adds the WebSocket as subscribed to that mailbox, so new messages are delivered immediately to the connected client. There is no explicit ack to the `open` command, but since all clients add a message to the mailbox as soon as they connect, there will always be a `message` response shortly after the `open` goes through. The `close` command provokes a `closed` response.

The `close` command accepts an optional "mood" string: this allows clients to tell the server (in general terms) about their experiences with the wormhole interaction. The server records the mood in its "usage" record, so the server operator can get a sense of how many connections are succeeding and failing. The moods currently recognized by the Mailbox Server are:

- `happy` (default): the PAKE key-establishment worked, and the client saw at least one valid encrypted message from its peer
- `lonely`: the client gave up without hearing anything from its peer
- `scary`: the client saw an invalid encrypted message from its peer, indicating that either the wormhole code was typed in wrong, or an attacker tried (and failed) to guess the code
- `errory`: the client encountered some other error: protocol problem or internal error

The server will also record `pruney` if it deleted the mailbox due to inactivity, or `crowded` if more than two sides tried to access the mailbox.

When clients use the `add` command to add a client-to-client message, they will put the body (a bytestring) into the command as a hex-encoded string in the `body` key. They will also put the message's "phase", as a string, into the `phase` key. See client-protocol.md for details about how different phases are used.

When a client sends `open`, it will get back a `message` response for every message in the mailbox. It will also get a real-time `message` for every `add` performed by clients later. These `message` responses include "side" and "phase" from the sending client, and "body" (as a hex string, encoding the binary message body). The decoded "body" will either by a random-looking cryptographic value (for the PAKE message), or a random-looking encrypted blob (for the VERSION message, as well as all application-provided payloads). The `message` response will also include `id`, copied from the `id` of the `add` message (and used only by the timing-diagram tool).

The Mailbox Server does not de-duplicate messages, nor does it retain ordering: clients must do both if they need to.

7.7 All Message Types

This lists all message types, along with the type-specific keys for each (if any), and which ones provoke direct responses:

- S->C welcome {welcome:}
- (C->S) bind {appid:, side:}
- (C->S) list {} -> nameplates
- S->C nameplates {nameplates: [{id: str},..]}
- (C->S) allocate {} -> allocated
- S->C allocated {nameplate:}
- (C->S) claim {nameplate:} -> claimed
- S->C claimed {mailbox:}
- (C->S) release {nameplate:?} -> released
- S->C released
- (C->S) open {mailbox:}
- (C->S) add {phase: str, body: hex} -> message (to all connected clients)
- S->C message {side:, phase:, body:, id:}
- (C->S) close {mailbox:?, mood:?} -> closed
- S->C closed
- S->C ack
- (C->S) ping {ping: int} -> ping
- S->C pong {pong: int}
- S->C error {error: str, orig:}

7.8 Persistence

The server stores all messages in a database, so it should not lose any information when it is restarted. The server will not send a direct response until any side-effects (such as the message being added to the mailbox) have been safely committed to the database.

The client library knows how to resume the protocol after a reconnection event, assuming the client process itself continues to run.

Clients which terminate entirely between messages (e.g. a secure chat application, which requires multiple wormhole messages to exchange address-book entries, and which must function even if the two apps are never both running at the same time) can use “Journal Mode” to ensure forward progress is made: see “journal.md” for details.

CHAPTER 8

Client-to-Client Protocol

Wormhole clients do not talk directly to each other (at least at first): they only connect directly to the Mailbox Server. They ask this server to convey messages to the other client (via the `add` command and the `message` response). This document explains the format of these client-to-client messages.

Each such message contains a "phase" string, and a hex-encoded binary "body".

Any phase which is purely numeric (`^\d+$`) is reserved for encrypted application data. The Mailbox server may deliver these messages multiple times, or out-of-order, but the wormhole client will deliver the corresponding decrypted data to the application in strict numeric order. All other (non-numeric) phases are reserved for the Wormhole client itself. Clients will ignore any phase they do not recognize.

Immediately upon opening the mailbox, clients send the `pake` phase, which contains the binary SPAKE2 message (the one computed as `X+M*pw` or `Y+N*pw`).

Upon receiving their peer's `pake` phase, clients compute and remember the shared key. They derive the "verifier" (a hash of the shared key) and deliver it to the application by calling `got_verifier`: applications can display this to users who want additional assurance (by manually comparing the values from both sides: they ought to be identical). At this point clients also send the encrypted `version` phase, whose plaintext payload is a UTF-8-encoded JSON-encoded dictionary of metadata. This allows the two Wormhole instances to signal their ability to do other things (like "dilate" the wormhole). The version data will also include an `app_versions` key which contains a dictionary of metadata provided by the application, allowing apps to perform similar negotiation.

At this stage, the client knows the supposed shared key, but has not yet seen evidence that the peer knows it too. When the first peer message arrives (i.e. the first message with a `.side` that does not equal our own), it will be decrypted: we use authenticated encryption (`nacl.SecretBox`), so if this decryption succeeds, then we're confident that *somebody* used the same wormhole code as us. This event pushes the client mood from "lonely" to "happy".

This might be triggered by the peer's `version` message, but if we had to re-establish the Mailbox Server connection, we might get peer messages out of order and see some application-level message first.

When a `version` message is successfully decrypted, the application is signaled with `got_version`. When any application message is successfully decrypted, `received` is signaled. Application messages are delivered strictly in-order: if we see phases 3 then 2 then 1, all three will be delivered in sequence after phase 1 is received.

If any message cannot be successfully decrypted, the mood is set to "scary", and the wormhole is closed. All pending Deferreds will be errbacked with a `WrongPasswordError` (a subclass of `WormholeError`), the nameplate/mailbox will be released, and the WebSocket connection will be dropped. If the application calls `close()`, the resulting Deferred will not fire until deallocation has finished and the WebSocket is closed, and then it will fire with an errback.

Both `version` and all numeric (app-specific) phases are encrypted. The message body will be the hex-encoded output of a NaCl `SecretBox`, keyed by a phase+side -specific key (computed with HKDF-SHA256, using the shared PAKE key as the secret input, and `wormhole:phase:%s%s % (SHA256(side), SHA256(phase))` as the CTXinfo), with a random nonce.

CHAPTER 9

File-Transfer Protocol

The `bin/wormhole` tool uses a Wormhole to establish a connection, then speaks a file-transfer -specific protocol over that Wormhole to decide how to transfer the data. This application-layer protocol is described here.

All application-level messages are dictionaries, which are JSON-encoded and and UTF-8 encoded before being handed to `wormhole.send` (which then encrypts them before sending through the mailbox server to the peer).

9.1 Sender

`wormhole send` has two main modes: file/directory (which requires a non-wormhole Transit connection), or text (which does not).

If the sender is doing files or directories, its first message contains just a `transit` key, whose value is a dictionary with `abilities-v1` and `hints-v1` keys. These are given to the Transit object, described below.

Then (for both files/directories and text) it sends a message with an `offer` key. The offer contains a single key, exactly one of (`message`, `file`, or `directory`). For `message`, the value is the message being sent. For `file` and `directory`, it contains a dictionary with additional information:

- `message`: the text message, for text-mode
- `file`: for file-mode, a dict with `filename` and `filesize`
- `directory`: for directory-mode, a dict with:
- `mode`: the compression mode, currently always `zipfile/deflated`
- `dirname`
- `zipsize`: integer, size of the transmitted data in bytes
- `numbytes`: integer, estimated total size of the uncompressed directory
- `numfiles`: integer, number of files+directories being sent

The sender runs a loop where it waits for similar dictionary-shaped messages from the recipient, and processes them. It reacts to the following keys:

- `error`: use the value to throw a TransferError and terminates
- `transit`: use the value to build the Transit instance
- `answer`:
- if `message_ack:  ok` is in the value (we're in text-mode), then exit with success
- if `file_ack:  ok` in the value (and we're in file/directory mode), then wait for Transit to connect, then send the file through Transit, then wait for an ack (via Transit), then exit

The sender can handle all of these keys in the same message, or spaced out over multiple ones. It will ignore any keys it doesn't recognize, and will completely ignore messages that don't contain any recognized key. The only constraint is that the message containing `message_ack` or `file_ack` is the last one: it will stop looking for wormhole messages at that point.

9.2 Recipient

`wormhole receive` is used for both file/directory-mode and text-mode: it learns which is being used from the `offer` message.

The recipient enters a loop where it processes the following keys from each received message:

- `error`: if present in any message, the recipient raises TransferError (with the value) and exits immediately (before processing any other keys)
- `transit`: the value is used to build the Transit instance
- `offer`: parse the offer:
- `message`: accept the message and terminate
- `file`: connect a Transit instance, wait for it to deliver the indicated number of bytes, then write them to the target filename
- `directory`: as with `file`, but unzip the bytes into the target directory

9.3 Transit

The Wormhole API does not currently provide for large-volume data transfer (this feature will be added to a future version, under the name "Dilated Wormhole"). For now, bulk data is sent through a "Transit" object, which does not use the Mailbox Server. Instead, it tries to establish a direct TCP connection from sender to recipient (or vice versa). If that fails, both sides connect to a "Transit Relay", a very simple Server that just glues two TCP sockets together when asked.

The Transit object is created with a key (the same key on each side), and all data sent through it will be encrypted with a derivation of that key. The transit key is also used to derive handshake messages which are used to make sure we're talking to the right peer, and to help the Transit Relay match up the two client connections. Unlike Wormhole objects (which are symmetric), Transit objects come in pairs: one side is the Sender, and the other is the Receiver.

Like Wormhole, Transit provides an encrypted record pipe. If you call `.send()` with 40 bytes, the other end will see a `.gotData()` with exactly 40 bytes: no splitting, merging, dropping, or re-ordering. The Transit object also functions as a twisted Producer/Consumer, so it can be connected directly to file-readers and writers, and does flow-control properly.

Most of the complexity of the Transit object has to do with negotiating and scheduling likely targets for the TCP connection.

Each Transit object has a set of "abilities". These are outbound connection mechanisms that the client is capable of using. The basic CLI tool (running on a normal computer) has two abilities: `direct-tcp-v1` and `relay-v1`.

- `direct-tcp-v1` indicates that it can make outbound TCP connections to a requested host and port number. "v1" means that the first thing sent over these connections is a specific derived handshake message, e.g. `transit sender HEXHEX ready\n\n`.
- `relay-v1` indicates it can connect to the Transit Relay and speak the matching protocol (in which the first message is `please relay HEXHEX for side HEX\n`, and the relay might eventually say `ok\n`).

Future implementations may have additional abilities, such as connecting directly to Tor onion services, I2P services, WebSockets, WebRTC, or other connection technologies. Implementations on some platforms (such as web browsers) may lack `direct-tcp-v1` or `relay-v1`.

While it isn't strictly necessary for both sides to emit what they're capable of using, it does help performance: a Tor Onion-service -capable receiver shouldn't spend the time and energy to set up an onion service if the sender can't use it.

After learning the abilities of its peer, the Transit object can create a list of "hints", which are endpoints that the peer should try to connect to. Each hint will fall under one of the abilities that the peer indicated it could use. Hints have types like `direct-tcp-v1`, `tor-tcp-v1`, and `relay-v1`. Hints are encoded into dictionaries (with a mandatory `type` key, and other keys as necessary):

- `direct-tcp-v1` {hostname:, port:, priority:?}
- `tor-tcp-v1` {hostname:, port:, priority:?}
- `relay-v1` {hints: [{hostname:, port:, priority:?}, ..]}

For example, if our peer can use `direct-tcp-v1`, then our Transit object will deduce our local IP addresses (unless forbidden, i.e. we're using Tor), listen on a TCP port, then send a list of `direct-tcp-v1` hints pointing at all of them. If our peer can use `relay-v1`, then we'll connect to our relay server and give the peer a hint to the same.

`tor-tcp-v1` hints indicate an Onion service, which cannot be reached without Tor. `direct-tcp-v1` hints can be reached with direct TCP connections (unless forbidden) or by proxying through Tor. Onion services take about 30 seconds to spin up, but bypass NAT, allowing two clients behind NAT boxes to connect without a transit relay (really, the entire Tor network is acting as a relay).

The file-transfer application uses `transit` messages to convey these abilities and hints from one Transit object to the other. After updating the Transit objects, it then asks the Transit object to connect, whereupon Transit will try to connect to all the hints that it can, and will use the first one that succeeds.

The file-transfer application, when actually sending file/directory data, will close the Wormhole as soon as it has enough information to begin opening the Transit connection. The final ack of the received data is sent through the Transit object, as a UTF-8-encoded JSON-encoded dictionary with `ack: ok` and `sha256: HEXHEX` containing the hash of the received data.

9.4 Future Extensions

Transit will be extended to provide other connection techniques:

- WebSocket: usable by web browsers, not too hard to use by normal computers, requires direct (or relayed) TCP connection
- WebRTC: usable by web browsers, hard-but-technically-possible to use by normal computers, provides NAT hole-punching for "free"
- (web browsers cannot make direct TCP connections, so interop between browsers and CLI clients will either require adding WebSocket to CLI, or a relay that is capable of speaking/bridging both)

- I2P: like Tor, but not capable of proxying to normal TCP hints.
- ICE-mediated STUN/STUNT: NAT hole-punching, assisted somewhat by a server that can tell you your external IP address and port. Maybe implemented as a uTP stream (which is UDP based, and thus easier to get through NAT).

The file-transfer protocol will be extended too:

- "command mode": establish the connection, *then* figure out what we want to use it for, allowing multiple files to be exchanged, in either direction. This is to support a GUI that lets you open the wormhole, then drop files into it on either end.
- some Transit messages being sent early, so ports and Onion services can be spun up earlier, to reduce overall waiting time
- transit messages being sent in multiple phases: maybe the transit connection can progress while waiting for the user to confirm the transfer

The hope is that by sending everything in dictionaries and multiple messages, there will be enough wiggle room to make these extensions in a backwards-compatible way. For example, to add "command mode" while allowing the fancy new (as yet unwritten) GUI client to interoperate with old-fashioned one-file-only CLI clients, we need the GUI tool to send an "I'm capable of command mode" in the VERSION message, and look for it in the received VERSION. If it isn't present, it will either expect to see an offer (if the other side is sending), or nothing (if it is waiting to receive), and can explain the situation to the user accordingly. It might show a locked set of bars over the wormhole graphic to mean "cannot send", or a "waiting to send them a file" overlay for send-only.

CHAPTER 10

Known Vulnerabilities

10.1 Low-probability Man-In-The-Middle Attacks

By default, wormhole codes contain 16 bits of entropy. If an attacker can intercept your network connection (either by owning your network, or owning the mailbox server), they can attempt an attack. They will have a one-in-65536 chance of successfully guessing your code, allowing them to pose as your intended partner. If they succeed, they can turn around and immediately start a new wormhole (using the same code), allowing your partner to connect to them instead of you. By passing, observing, and possibly modifying messages between these two connections, they could perform an MitM (Man In The Middle) attack.

If the server refused to re-use the same channel id (aka “nameplate”) right away (issue #31), a network attacker would be unable to set up the second connection, cutting this attack in half. An attacker who controls the server would not be affected.

Basic probability tells us that peers will see a large number of WrongPasswordErrors before the attacker has a useful chance of successfully guessing any wormhole code. You should expect to see about 32000 failures before they have a 50% chance of being successful. If you see many failures, and think someone is trying to guess your codes, you can use e.g. `wormhole send --code-length=4` to make a longer code (reducing their chances significantly).

Of course, an attacker who learns your secret wormhole code directly (because you delivered it over an insecure channel) can perform this attack with 100% reliability.

10.2 DoS Attack on the Mailbox Server

Wormhole codes can be so short because they implicitly contain a common mailbox server URL (any two applications that use magic-wormhole should be configured to use the same server). As a result, successful operation depends upon both clients being able to contact that server, making it a SPOF (single point of failure).

In particular, grumpy people could disrupt service for everyone by writing a program that just keeps connecting to the mailbox server, pretending to be real clients, and claiming messages meant for legitimate users.

I do not have any good mitigations for this attack, and functionality may depend upon the continued goodwill of potential vandals. The weak ones that I’ve considered (but haven’t implemented yet) include:

- hashcash challenges when the server is under attack
- per-IP rate-limiting (although I'd want to be careful about protecting the privacy of the IP addresses, so it'd need a rotating hash seed)
- require users to go through some external service (maybe ReCAPTCHA?) and get a rate-limiting ticket before claiming a channel
- shipping an attack tool (flooding the first million channels), as part of the distribution, in a subcommand named `wormhole break-this-useful-service-for-everybody-because-i-am-a-horrible-person`, in the hopes that pointing out how easy it is might dissuade a few would-be vandals from feeling a sense of accomplishment at writing their own :). Not sure it would help much, but I vaguely remember hearing about something similar in the early multi-user unix systems (a publicly-executable /bin/crash or something, which new users tended to only run once before learning some responsibility).

Using the secret words as part of the "channel id" isn't safe, since it would allow a network attacker, or the mailbox server, to deduce what the secret words are: since they only have 16 bits of entropy, the attacker just makes a table of hash(words) -> channel-id, then reverses it. To make that safer we'd need to increase the codes to maybe 80 bits (ten words), plus do some significant key-stretching (like 5-10 seconds of scrypt or argon2), which would increase latency and CPU demands, and still be less secure overall.

The core problem is that, because things are so easy for the legitimate participants, they're really easy for the attacker too. Short wormhole codes are the easiest to use, but they make it for a trivially predictable channel-id target.

I don't have a good answer for this one. I'm hoping that it isn't sufficiently interesting to attack that it'll be an issue, but I can't think of any simple answers. If the API is sufficiently compelling for other applications to incorporate Wormhole "technology" into their apps, I'm expecting that they'll run their own mailbox server, and of course those apps can incorporate whatever sort of DoS protection seems appropriate. For the built-in/upstream send-text/file/directory tools, using the public relay that I run, it may just have to be a best-effort service, and if someone decides to kill it, it fails.

See #107 for more discussion.

CHAPTER 11

Journaled Mode

(note: this section is speculative, the code has not yet been written)

Magic-Wormhole supports applications which are written in a "journaled" or "checkpointed" style. These apps store their entire state in a well-defined checkpoint (perhaps in a database), and react to inbound events or messages by carefully moving from one state to another, then releasing any outbound messages. As a result, they can be terminated safely at any moment, without warning, and ensure that the externally-visible behavior is deterministic and independent of this stop/restart timing.

This is the style encouraged by the E event loop, the original Waterken Server, and the more modern Ken Platform, all influential in the object-capability security community.

11.1 Requirements

Applications written in this style must follow some strict rules:

- all state goes into the checkpoint
- the only way to affect the state is by processing an input message
- event processing is deterministic (any non-determinism must be implemented as a message, e.g. from a clock service or a random-number generator)
- apps must never forget a message for which they've accepted responsibility

The main processing function takes the previous state checkpoint and a single input message, and produces a new state checkpoint and a set of output messages. For performance, the state might be kept in memory between events, but the behavior should be indistinguishable from that of a server which terminates completely between events.

In general, applications must tolerate duplicate inbound messages, and should re-send outbound messages until the recipient acknowledges them. Any outbound responses to an inbound message must be queued until the checkpoint is recorded. If outbound messages were delivered before the checkpointing, then a crash just after delivery would roll the process back to a state where it forgot about the inbound event, causing observably inconsistent behavior that depends upon whether the outbound message successfully escaped the dying process or not.

As a result, journaled-style applications use a very specific process when interacting with the outside world. Their event-processing function looks like:

- receive inbound event
- (load state)
- create queue for any outbound messages
- process message (changing state and queuing outbound messages)
- serialize state, record in checkpoint
- deliver any queued outbound messages

In addition, the protocols used to exchange messages should include message IDs and acks. Part of the state vector will include a set of unacknowledged outbound messages. When a connection is established, all outbound messages should be re-sent, and messages are removed from the pending set when an inbound ack is received. The state must include a set of inbound message ids which have been processed already. All inbound messages receive an ack, but only new ones are processed. Connection establishment/loss is not strictly included in the journaled-app model (in Waterken/Ken, message delivery is provided by the platform, and apps do not know about connections), but general:

- "I want to have a connection" is stored in the state vector
- "I am connected" is not
- when a connection is established, code can run to deliver pending messages, and this does not qualify as an inbound event
- inbound events can only happen when at least one connection is established
- immediately after restarting from a checkpoint, no connections are established, but the app might initiate outbound connections, or prepare to accept inbound ones

11.2 Wormhole Support

To support this mode, the Wormhole constructor accepts a `journal=` argument. If provided, it must be an object that implements the `wormhole.IJournal` interface, which consists of two methods:

- `j.queue_outbound(fn, *args, **kwargs)`: used to delay delivery of outbound messages until the checkpoint has been recorded
- `j.process()`: a context manager which should be entered before processing inbound messages

`wormhole.Journal` is an implementation of this interface, which is constructed with a (synchronous) `save_checkpoint` function. Applications can use it, or bring their own.

The Wormhole object, when configured with a journal, will wrap all inbound WebSocket message processing with the `j.process()` context manager, and will deliver all outbound messages through `j.queue_outbound`. Applications using such a Wormhole must also use the same journal for their own (non-wormhole) events. It is important to coordinate multiple sources of events: e.g. a UI event may cause the application to call `w.send(data)`, and the outbound wormhole message should be checkpointed along with the app's state changes caused by the UI event. Using a shared journal for both wormhole- and non-wormhole- events provides this coordination.

The `save_checkpoint` function should serialize application state along with any Wormholes that are active. Wormhole state can be obtained by calling `w.serialize()`, which will return a dictionary (that can be JSON-serialized). At application startup (or checkpoint resumption), Wormholes can be regenerated with `wormhole.from_serialized()`. Note that only "delegated-mode" wormholes can be serialized: Deferreds are not amenable to usage beyond a single process lifetime.

For a functioning example of a journaled-mode application, see misc/demo-journal.py. The following snippet may help illustrate the concepts:

```python
class App:
    @classmethod
    def new(klass):
        self = klass()
        self.state = {}
        self.j = wormhole.Journal(self.save_checkpoint)
        self.w = wormhole.create(.., delegate=self, journal=self.j)

    @classmethod
    def from_serialized(klass):
        self = klass()
        self.j = wormhole.Journal(self.save_checkpoint)
        with open("state.json", "r") as f:
            data = json.load(f)
        self.state = data["state"]
        self.w = wormhole.from_serialized(data["wormhole"], reactor,
                                          delegate=self, journal=self.j)

    def inbound_event(self, event):
        # non-wormhole events must be performed in the journal context
        with self.j.process():
            parse_event(event)
            change_state()
            self.j.queue_outbound(self.send, outbound_message)

    def wormhole_received(self, data):
        # wormhole events are already performed in the journal context
        change_state()
        self.j.queue_outbound(self.send, stuff)

    def send(self, outbound_message):
        actually_send_message(outbound_message)

    def save_checkpoint(self):
        app_state = {"state": self.state, "wormhole": self.w.serialize()}
        with open("state.json", "w") as f:
            json.dump(app_state, f)
```

CHAPTER 12

Indices and tables

- genindex
- modindex
- search