
Reticulum Network Stack

Release 0.3.5 beta

Mark Qvist

Apr 28, 2022

CONTENTS

1	What is Reticulum?	3
1.1	Current Status	3
1.2	What does Reticulum Offer?	3
1.3	Where can Reticulum be Used?	4
1.4	Interface Types and Devices	4
1.5	Caveat Emptor	5
2	Getting Started Fast	7
2.1	Try Using a Reticulum-based Program	7
2.1.1	Nomad Network	7
2.1.2	Sideband	8
2.2	Using the Included Utilities	10
2.3	Creating a Network With Reticulum	10
2.4	Connecting Reticulum Instances Over the Internet	10
2.5	Connect to the Public Testnet	11
2.6	Develop a Program with Reticulum	11
2.7	Participate in Reticulum Development	12
2.8	Reticulum on ARM64	12
2.9	Reticulum on Android	13
2.10	Adding Radio Interfaces	13
3	Using Reticulum on Your System	15
3.1	Included Utility Programs	15
3.1.1	The rnsd Utility	15
3.1.2	The rnstatus Utility	16
3.1.3	The rnpath Utility	17
3.1.4	The rnprobe Utility	17
3.2	Improving System Configuration	18
3.2.1	Fixed Serial Port Names	18
3.2.2	Reticulum as a System Service	19
4	Building Networks	21
4.1	Concepts & Overview	21
4.2	Example Scenarios	22
4.2.1	Interconnected LoRa Sites	22
4.2.2	Bridging Over the Internet	23
4.2.3	Growth and Convergence	23
5	Supported Interfaces	25
5.1	Common Interface Options	25

5.2	Auto Interface	26
5.3	I2P Interface	27
5.4	TCP Server Interface	28
5.5	TCP Client Interface	28
5.6	UDP Interface	29
5.7	RNode LoRa Interface	30
5.8	Serial Interface	31
5.9	KISS Interface	32
5.10	AX.25 KISS Interface	33
6	Understanding Reticulum	35
6.1	Motivation	35
6.2	Goals	36
6.3	Introduction & Basic Functionality	36
6.3.1	Destinations	37
6.3.2	Public Key Announcements	39
6.3.3	Identities	39
6.3.4	Getting Further	40
6.4	Reticulum Transport	40
6.4.1	Node Types	40
6.4.2	The Announce Mechanism in Detail	40
6.4.3	Reaching the Destination	41
6.4.4	Resources	43
6.5	Reference Setup	44
6.6	Protocol Specifics	45
6.6.1	Packet Prioritisation	45
6.6.2	Interface Access Codes	45
6.6.3	Wire Format	45
7	API Reference	49
7.1	Classes	49
7.1.1	Reticulum	49
7.1.2	Identity	50
7.1.3	Destination	52
7.1.4	Packet	55
7.1.5	Packet Receipt	55
7.1.6	Link	56
7.1.7	Request Receipt	58
7.1.8	Resource	58
7.1.9	Transport	59
8	Code Examples	61
8.1	Minimal	61
8.2	Announce	63
8.3	Broadcast	67
8.4	Echo	69
8.5	Link	76
8.6	Identification	81
8.7	Requests & Responses	88
8.8	Filetransfer	93
	Index	107

This manual aims to provide you with all the information you need to understand Reticulum, build networks or develop programs using it, or to participate in the development of Reticulum itself.

WHAT IS RETICULUM?

Reticulum is a cryptography-based networking stack for building wide-area networks with readily available hardware, that can continue to operate even with extremely low bandwidth and very high latency.

Reticulum allows you to build wide-area networks with off-the-shelf tools, and offers end-to-end encryption, autoconfiguring cryptographically backed multi-hop transport, efficient addressing, unforgeable packet acknowledgements and more.

Reticulum is a complete networking stack, and does not need IP or higher layers, although it is easy to utilise IP (with TCP or UDP) as the underlying carrier for Reticulum. It is therefore trivial to tunnel Reticulum over the Internet or private IP networks. Reticulum is built directly on cryptographic principles, allowing resilience and stable functionality in open and trustless networks.

No kernel modules or drivers are required. Reticulum runs completely in userland, and can run on practically any system that runs Python 3. Reticulum runs well even on small single-board computers like the Pi Zero.

1.1 Current Status

Reticulum should currently be considered beta software. All core protocol features are implemented and functioning, but additions will probably occur as real-world use is explored. There will be bugs. The API and wire-format can be considered stable at the moment, but could change if absolutely warranted.

1.2 What does Reticulum Offer?

- Coordination-less globally unique addressing and identification
- Fully self-configuring multi-hop routing
- Complete initiator anonymity, communicate without revealing your identity
- Asymmetric encryption based on X25519, and Ed25519 signatures as a basis for all communication
- Forward Secrecy by using ephemeral Elliptic Curve Diffie-Hellman keys on Curve25519
- Reticulum uses the [Fernet](#) specification for on-the-wire / over-the-air encryption
 - All keys are ephemeral and derived from an ECDH key exchange on Curve25519
 - AES-128 in CBC mode with PKCS7 padding
 - HMAC using SHA256 for authentication
 - IVs are generated through `os.urandom()`
- Unforgeable packet delivery confirmations

- A variety of supported interface types
- An intuitive and developer-friendly API
- Efficient link establishment
 - Total bandwidth cost of setting up a link is only 3 packets, totalling 237 bytes
 - Low cost of keeping links open at only 0.62 bits per second
- Reliable and efficient transfer of arbitrary amounts of data
 - Reticulum can handle a few bytes of data or files of many gigabytes
 - Sequencing, transfer coordination and checksumming is automatic
 - The API is very easy to use, and provides transfer progress
- Authentication and virtual network segmentation on all supported interface types
- Flexible scalability allowing extremely low-bandwidth networks to co-exist and interoperate with large, high-bandwidth networks

1.3 Where can Reticulum be Used?

Over practically any medium that can support at least a half-duplex channel with 500 bits per second throughput, and an MTU of 500 bytes. Data radios, modems, LoRa radios, serial lines, AX.25 TNCs, amateur radio digital modes, ad-hoc WiFi, free-space optical links and similar systems are all examples of the types of interfaces Reticulum was designed for.

An open-source LoRa-based interface called [RNode](#) has been designed as an example transceiver that is very suitable for Reticulum. It is possible to build it yourself, to transform a common LoRa development board into one, or it can be purchased as a complete transceiver.

Reticulum can also be encapsulated over existing IP networks, so there's nothing stopping you from using it over wired ethernet or your local WiFi network, where it'll work just as well. In fact, one of the strengths of Reticulum is how easily it allows you to connect different mediums into a self-configuring, resilient and encrypted mesh.

As an example, it's possible to set up a Raspberry Pi connected to both a LoRa radio, a packet radio TNC and a WiFi network. Once the interfaces are added, Reticulum will take care of the rest, and any device on the WiFi network can communicate with nodes on the LoRa and packet radio sides of the network, and vice versa.

1.4 Interface Types and Devices

Reticulum implements a range of generalised interface types that covers the communications hardware that Reticulum can run over. If your hardware is not supported, it's relatively simple to implement an interface class. Currently, Reticulum can use the following devices and communication mediums:

- Any ethernet device
 - WiFi devices
 - Wired ethernet devices
 - Fibre-optic transceivers
 - Data radios with ethernet ports
- LoRa using [RNode](#)
 - Can be installed on [many popular LoRa boards](#)

- Can be purchased as a [ready to use transceiver](#)
- Packet Radio TNCs, such as [OpenModem](#)
 - Any packet radio TNC in KISS mode
 - Ideal for VHF and UHF radio
- Any device with a serial port
- The I2P network
- TCP over IP networks
- UDP over IP networks

For a full list and more details, see the [Supported Interfaces](#) chapter.

1.5 Caveat Emptor

Reticulum is an experimental networking stack, and should be considered as such. While it has been built with cryptography best-practices very foremost in mind, it has not been externally security audited, and there could very well be privacy-breaking bugs. To be considered secure, Reticulum needs a thorough security review by independent cryptographers and security researchers. If you want to help out, or help sponsor an audit, please do get in touch.

GETTING STARTED FAST

The best way to get started with the Reticulum Network Stack depends on what you want to do. This guide will outline sensible starting paths for different scenarios.

2.1 Try Using a Reticulum-based Program

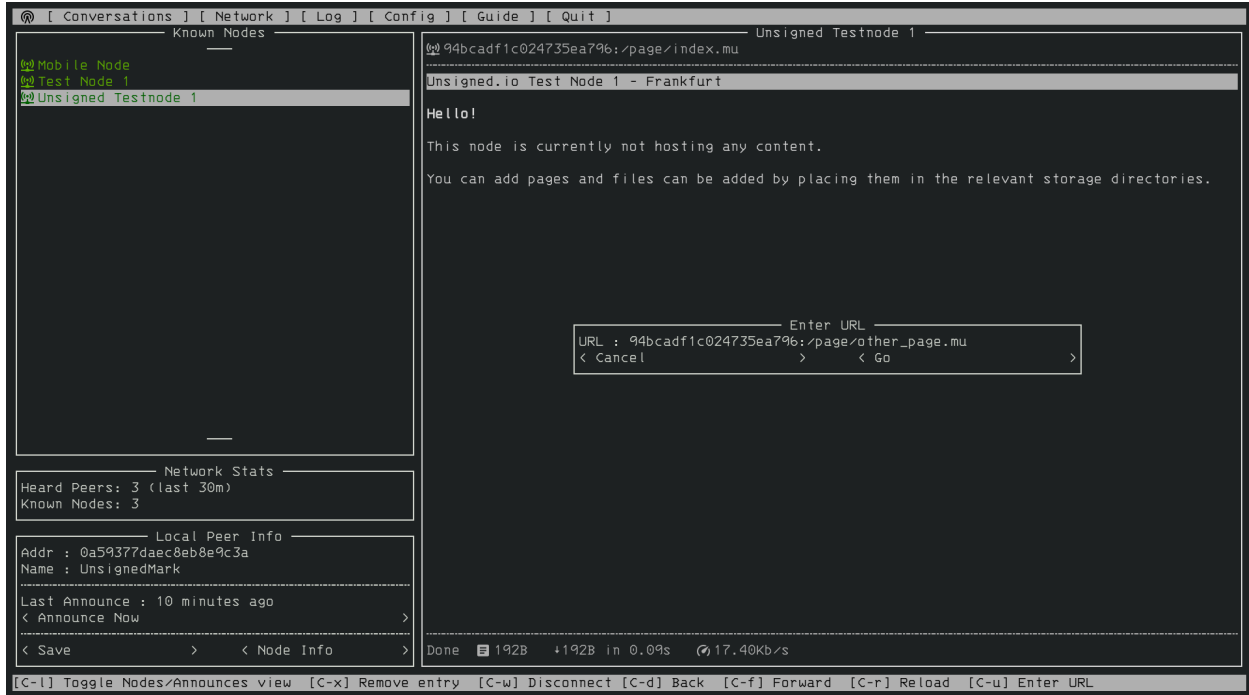
If you simply want to try using a program built with Reticulum, a few different programs exist that allow basic communication and a range of other useful functions over even extremely low-bandwidth Reticulum networks.

These programs will let you get a feel for how Reticulum works. They have been designed to run well over networks based on LoRa or packet radio, but can also be used completely over local WiFi, wired ethernet, the Internet, or any combination.

As such, it is easy to get started experimenting, without having to set up any radio transceivers or infrastructure just to try it out. Launching the programs on separate devices connected to the same WiFi network is enough to get started, and physical radio interfaces can then be added later.

2.1.1 Nomad Network

The terminal-based program [Nomad Network](#) provides a complete encrypted communications suite built with Reticulum. It features encrypted messaging (both direct and delayed-delivery for offline users), file sharing, and has a built-in text-browser and page server with support for dynamically rendered pages, user authentication and more.



Nomad Network is a user-facing client for the messaging and information-sharing protocol [LXMF](#), another project built with Reticulum.

You can install Nomad Network via pip:

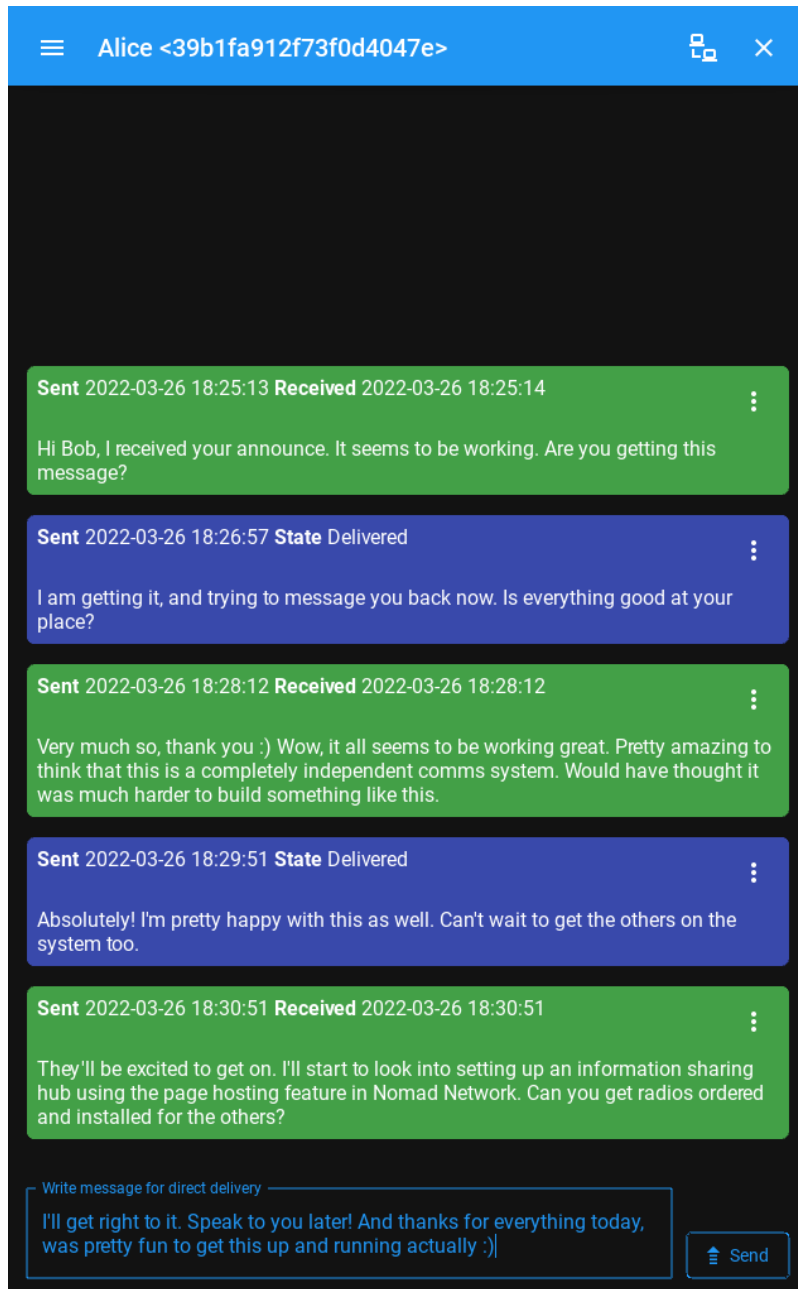
```
# Install ...
pip3 install nomadnet

# ... and run
nomadnet
```

Please Note: If this is the very first time you use pip to install a program on your system, you might need to reboot your system for your program to become available. If you get a “command not found” error or similar when running the program, reboot your system and try again.

2.1.2 Sideband

If you would rather use a program with a graphical user interface, you can take a look at [Sideband](#), which is available for Android, Linux and macOS.



Sideband is currently in the early stages of development, but already provides basic communication features, and interoperates with Nomad Network, or any other LXMF client.

2.2 Using the Included Utilities

Reticulum comes with a range of included utilities that make it easier to manage your network, check connectivity and make Reticulum available to other programs on your system.

You can use `rnsd` to run Reticulum as a background or foreground service, and the `rnsstatus`, `rnsdpath` and `rnsdprobe` utilities to view and query network status and connectivity.

To learn more about these utility programs, have a look at the *Using Reticulum on Your System* chapter of this manual.

2.3 Creating a Network With Reticulum

To create a network, you will need to specify one or more *interfaces* for Reticulum to use. This is done in the Reticulum configuration file, which by default is located at `~/.reticulum/config`. You can edit this file by hand, or use the interactive `rnsconfig` utility.

When Reticulum is started for the first time, it will create a default configuration file, with one active interface. This default interface uses your existing ethernet and WiFi networks (if any), and only allows you to communicate with other Reticulum peers within your local broadcast domains.

To communicate further, you will have to add one or more interfaces. The default configuration includes a number of examples, ranging from using TCP over the internet, to LoRa and Packet Radio interfaces.

With Reticulum, you only need to configure what interfaces you want to communicate over. There is no need to configure address spaces, subnets, routing tables, or other things you might be used to from other network types.

Once Reticulum knows which interfaces it should use, it will automatically discover topography and configure transport of data to any destinations it knows about.

In situations where you already have an established WiFi or ethernet network, and many devices that want to utilise the same external Reticulum network (for example over LoRa), it will often be sufficient to let one system act as a Reticulum gateway, by adding any external interfaces to this systems configuration, and enabling transport. Any other device on your local WiFi will then be able to connect to this wider Reticulum network just using the default interface configuration.

Possibly, the examples in the config file are enough to get you started. If you want more information, you can read the *Building Networks* and *Interfaces* chapters of this manual.

2.4 Connecting Reticulum Instances Over the Internet

Reticulum currently offers two interfaces suitable for connecting instances over the Internet: *TCP* and *I2P*. Each interface offers a different set of features, and Reticulum users should carefully choose the interface which best suites their needs.

The `TCPServerInterface` allows users to host an instance accessible over TCP/IP. This method is generally faster, lower latency, and more energy efficient than using `I2PInterface`, however it also leaks more data about the server host.

TCP connections reveal the IP address of both your instance and the server to anyone who can inspect the connection. Someone could use this information to determine your location or identity. Adversaries inspecting your packets may be able to record packet metadata like time of transmission and packet size. Even though Reticulum encrypts traffic, TCP does not, so an adversary may be able to use packet inspection to learn that a system is running Reticulum, and what other IP addresses connect to it. Hosting a publicly reachable instance over TCP also requires a publicly reachable IP address, which most Internet connections don't offer anymore.

The `I2PInterface` routes messages through the [Invisible Internet Protocol \(I2P\)](#). To properly use this interface, users must also run an I2P daemon in parallel to `rnsd`. For always-on I2P nodes it is recommended to use `i2pd`.

By default, I2P will encrypt and mix all traffic sent over the Internet, and hide both the sender and receiver Reticulum instance IP addresses. Running an I2P node will also relay other I2P user's encrypted packets, which will use extra bandwidth and compute power, but also makes timing attacks and other forms of deep-packet-inspection much more difficult.

I2P also allows users to host globally available Reticulum instances from non-public IPs and behind firewalls.

In general it is recommended to use an I2P node if you want to host a publically accessible instance, while preserving anonymity. If you care more about performance, and a slightly easier setup, use TCP.

2.5 Connect to the Public Testnet

An experimental public testnet has been made accessible over both I2P and TCP. You can join it by adding one of the following interfaces to your `.reticulum/config` file:

```
# For connecting over TCP/IP:
[[RNS Testnet Frankfurt]]
  type = TCPClientInterface
  interface_enabled = yes
  outgoing = True
  target_host = frankfurt.rns.unsigned.io
  target_port = 4965

# For connecting over I2P:
[[RNS Testnet I2P Node A]]
  type = I2PInterface
  interface_enabled = yes
  peers = ykzlw5ujbaqc2xkec4cpvgyxj257wcrmmgkuxqmcur7cq3w3lha.b32.i2p
```

Many other Reticulum instances are connecting to this testnet, and you can also join it via other entry points if you know them. There is absolutely no control over the network topography, usage or what types of instances connect. It will also occasionally be used to test various failure scenarios, and there are no availability or service guarantees.

2.6 Develop a Program with Reticulum

If you want to develop programs that use Reticulum, the easiest way to get started is to install the latest release of Reticulum via `pip`:

```
pip3 install rns
```

The above command will install Reticulum and dependencies, and you will be ready to import and use RNS in your own programs. The next step will most likely be to look at some [Example Programs](#).

For extended functionality, you can install optional dependencies:

```
pip3 install pyserial netifaces
```

Further information can be found in the [API Reference](#).

2.7 Participate in Reticulum Development

If you want to participate in the development of Reticulum and associated utilities, you'll want to get the latest source from GitHub. In that case, don't use pip, but try this recipe:

```
# Install dependencies
pip3 install cryptography pyserial netifaces

# Clone repository
git clone https://github.com/markqvist/Reticulum.git

# Move into Reticulum folder and symlink library to examples folder
cd Reticulum
ln -s ../RNS ./Examples/

# Run an example
python3 Examples/Echo.py -s

# Unless you've manually created a config file, Reticulum will do so now,
# and immediately exit. Make any necessary changes to the file:
nano ~/.reticulum/config

# ... and launch the example again.
python3 Examples/Echo.py -s

# You can now repeat the process on another computer,
# and run the same example with -h to get command line options.
python3 Examples/Echo.py -h

# Run the example in client mode to "ping" the server.
# Replace the hash below with the actual destination hash of your server.
python3 Examples/Echo.py 3e12fc71692f8ec47bc5

# Have a look at another example
python3 Examples/Filetransfer.py -h
```

When you have experimented with the basic examples, it's time to go read the *Understanding Reticulum* chapter.

2.8 Reticulum on ARM64

On some architectures, including ARM64, not all dependencies have precompiled binaries. On such systems, you will need to install `python3-dev` before installing Reticulum or programs that depend on Reticulum.

```
# Install Python and development packages
sudo apt update
sudo apt install python3 python3-pip python3-dev

# Install Reticulum
python3 -m pip install rns
```


2.9 Reticulum on Android

Reticulum can be used on Android in different ways. The easiest way to get started is using an app like [Sideband](#).

For more control and features, you can use Reticulum and related programs via the [Termux](#) app, at the time of writing available on [F-droid](#).

Termux is a terminal emulator and Linux environment for Android based devices, which includes the ability to use many different programs and libraries, including Reticulum.

Since the Python `cryptography.io` module does not offer pre-built wheels for Android, the standard one-line install of Reticulum does not work on Android, and a few extra commands are required.

From within Termux, execute the following:

```
# First, make sure indexes and packages are up to date.
pkg update
pkg upgrade

# Then install dependencies for the cryptography library.
pkg install python build-essential openssl libffi rust

# Make sure pip is up to date, and install the wheel module.
pip3 install wheel pip --upgrade

# To allow the installer to build the cryptography module,
# we need to let it know what platform we are compiling for:
export CARGO_BUILD_TARGET="aarch64-linux-android"

# Start the install process for the cryptography module.
# Depending on your device, this can take several minutes,
# since the module must be compiled locally on your device.
pip3 install cryptography

# If the above installation succeeds, you can now install
# Reticulum and any related software
pip3 install rns
```

It is also possible to include Reticulum in apps compiled and distributed as Android APKs. A detailed tutorial and example source code will be included here at a later point.

2.10 Adding Radio Interfaces

Once you have Reticulum installed and working, you can add radio interfaces with any compatible hardware you have available. For information on how to configure this, see the [Interfaces](#) section of this manual.

A range of common LoRa development boards and transceiver modules can be used as interfaces with Reticulum. You can refer to the following external resources for more information:

- [How To Make Your Own RNodes](#)
- [Installing RNode Firmware on Compatible LoRa Devices](#)
- [Private, Secure and Uncensorable Messaging Over a LoRa Mesh](#)
- [RNode Firmware](#)

If you have communications hardware that you think would be suitable for use with Reticulum, you are welcome to head over to the [GitHub discussion pages](#) and propose adding an interface for the hardware.

USING RETICULUM ON YOUR SYSTEM

Reticulum is not installed as a driver or kernel module, as one might expect of a networking stack. Instead, Reticulum is distributed as a Python module. This means that no special privileges are required to install or use it. It is also very light-weight, and easy to transfer to and install on new systems. Any program or application that uses Reticulum will automatically load and initialise Reticulum when it starts.

In many cases, this approach is sufficient. When any program needs to use Reticulum, it is loaded, initialised, interfaces are brought up, and the program can now communicate over any Reticulum networks available. If another program starts up and also wants access to the same Reticulum network, the instance is simply shared. This works for any number of programs running concurrently, and is very easy to use, but depending on your use case, there are other options.

3.1 Included Utility Programs

If you often use Reticulum from several different programs, or simply want Reticulum to stay available all the time, for example if you are hosting a transport node, you might want to run Reticulum as a separate service that other programs, applications and services can utilise.

3.1.1 The `rnsd` Utility

To do so is very easy. Simply run the included `rnsd` command. When `rnsd` is running, it will keep all configured interfaces open, handle transport if it is enabled, and allow any other programs to immediately utilise the Reticulum network it is configured for.

You can even run multiple instances of `rnsd` with different configurations on the same system.

```
# Install Reticulum
pip3 install rns

# Run rnsd
rnsd
```

```
usage: rnsd [-h] [--config CONFIG] [-v] [-q] [--version]

Reticulum Network Stack Daemon

optional arguments:
  -h, --help            show this help message and exit
  --config CONFIG       path to alternative Reticulum config directory
  -v, --verbose
```

(continues on next page)

(continued from previous page)

```
-q, --quiet
--version      show program's version number and exit
```

You can easily add rnsd as an always-on service by *configuring a service*.

3.1.2 The rnsstatus Utility

Using the rnsstatus utility, you can view the status of configured Reticulum interfaces, similar to the ifconfig program.

```
# Run rnsstatus
rnsstatus

# Example output
Shared Instance[37428]
  Status : Up
  Serving : 1 program
  Rate   : 1.00 Gbps
  Traffic : 83.13 KB↑
           86.10 KB↓

AutoInterface[Local]
  Status : Up
  Mode   : Full
  Rate   : 10.00 Mbps
  Peers  : 1 reachable
  Traffic : 63.23 KB↑
           80.17 KB↓

TCPInterface[RNS Testnet Frankfurt/frankfurt.rns.unsigned.io:4965]
  Status : Up
  Mode   : Full
  Rate   : 10.00 Mbps
  Traffic : 187.27 KB↑
           74.17 KB↓

RNodeInterface[RNode UHF]
  Status : Up
  Mode   : Access Point
  Rate   : 1.30 kbps
  Access : 64-bit IFAC by <...e702c42ba8>
  Traffic : 8.49 KB↑
           9.23 KB↓

Reticulum Transport Instance <5245a8efe1788c6a70e1> running
```

```
usage: rnsstatus [-h] [--config CONFIG] [--version] [-a] [-v]
```

Reticulum Network Stack Status

optional arguments:

(continues on next page)

(continued from previous page)

```
-h, --help      show this help message and exit
--config CONFIG path to alternative Reticulum config directory
--version      show program's version number and exit
-a, --all      show all interfaces
-v, --verbose
```

3.1.3 The rnpath Utility

With the rnpath utility, you can look up and view paths for destinations on the Reticulum network.

```
# Run rnpath
rnpath eca6f4e4dc26ae329e61

# Example output
Path found, destination <eca6f4e4dc26ae329e61> is 4 hops away via <56b115c30cd386cad69c>_
↳on TCPInterface[Testnet/frankfurt.rns.unsigned.io:4965]
```

```
usage: rnpath [-h] [--config CONFIG] [--version] [-t] [-d] [-w seconds] [-v]
            [destination]
```

Reticulum Path Discovery Utility

positional arguments:

destination hexadecimal hash of the destination

optional arguments:

```
-h, --help      show this help message and exit
--config CONFIG path to alternative Reticulum config directory
--version      show program's version number and exit
-t, --table    show all known paths
-d, --drop     remove the path to a destination
-w seconds     timeout before giving up
-v, --verbose
```

3.1.4 The rnprobe Utility

The rnprobe utility lets you probe a destination for connectivity, similar to the ping program. Please note that probes will only be answered if the specified destination is configured to send proofs for received packets. Many destinations will not have this option enabled, and will not be probable.

```
# Run rnprobe
python3 -m RNS.Utilities.rnprobe example_utilities.echo.request 9382f334de63217a4278

# Example output
Sent 16 byte probe to <9382f334de63217a4278>
Valid reply received from <9382f334de63217a4278>
Round-trip time is 38.469 milliseconds over 2 hops
```

```
usage: rnprobe.py [-h] [--config CONFIG] [--version] [-v] [full_name] [destination_hash]
```

Reticulum Probe Utility

positional arguments:

```
  full_name      full destination name in dotted notation
  destination_hash hexadecimal hash of the destination
```

optional arguments:

```
-h, --help          show this help message and exit
--config CONFIG     path to alternative Reticulum config directory
--version           show program's version number and exit
-v, --verbose
```

3.2 Improving System Configuration

If you are setting up a system for permanent use with Reticulum, there is a few system configuration changes that can make this easier to administrate. These changes will be detailed here.

3.2.1 Fixed Serial Port Names

On a Reticulum instance with several serial port based interfaces, it can be beneficial to use the fixed device names for the serial ports, instead of the dynamically allocated shorthands such as `/dev/ttyUSB0`. Under most Debian-based distributions, including Ubuntu and Raspberry Pi OS, these nodes can be found under `/dev/serial/by-id`.

You can use such a device path directly in place of the numbered shorthands. Here is an example of a packet radio TNC configured as such:

```
[[Packet Radio KISS Interface]]
  type = KISSInterface
  interface_enabled = True
  outgoing = true
  port = /dev/serial/by-id/usb-FTDI_FT230X_Basic_UART_43891CKM-if00-port0
  speed = 115200
  databits = 8
  parity = none
  stopbits = 1
  preamble = 150
  txtail = 10
  persistence = 200
  slottime = 20
```

Using this methodology avoids potential naming mix-ups where physical devices might be plugged and unplugged in different orders, or when device name assignment varies from one boot to another.

3.2.2 Reticulum as a System Service

Instead of starting Reticulum manually, you can install `rnsd` as a system service and have it start automatically at boot.

If you installed Reticulum with `pip`, the `rnsd` program will most likely be located in a user-local installation path only, which means `systemd` will not be able to execute it. In this case, you can simply symlink the `rnsd` program into a directory that is in `systemd`'s path:

```
sudo ln -s $(which rnsd) /usr/local/bin/
```

You can then create the service file `/etc/systemd/system/rnsd.service` with the following content:

```
[Unit]
Description=Reticulum Network Stack Daemon
After=multi-user.target

[Service]
# If you run Reticulum on WiFi devices,
# or other devices that need some extra
# time to initialise, you might want to
# add a short delay before Reticulum is
# started by systemd:
# ExecStartPre=/bin/sleep 10
Type=simple
Restart=always
RestartSec=3
User=USERNAMEHERE
ExecStart=rnsd --service

[Install]
WantedBy=multi-user.target
```

Be sure to replace `USERNAMEHERE` with the user you want to run `rnsd` as.

To manually start `rnsd` run:

```
sudo systemctl start rnsd
```

If you want to automatically start `rnsd` at boot, run:

```
sudo systemctl enable rnsd
```


BUILDING NETWORKS

This chapter will provide you with the knowledge needed to build networks with Reticulum, which can often be easier than using traditional stacks, since you don't have to worry about coordinating addresses, subnets and routing for an entire network that you might not know how will evolve in the future. With Reticulum, you can simply add more segments to your network when it becomes necessary, and Reticulum will handle the convergence of the entire network automatically.

4.1 Concepts & Overview

There are important points that need to be kept in mind when building networks with Reticulum:

- In a Reticulum network, any node can autonomously generate as many addresses (called *destinations* in Reticulum terminology) as it needs, which become globally reachable to the rest of the network. There is no central point of control over the address space.
- Reticulum was designed to handle both very small, and very large networks. While the address space can support billions of endpoints, Reticulum is also very useful when just a few devices need to communicate.
- Low-bandwidth networks, like LoRa and packet radio, can interoperate and interconnect with much larger and higher bandwidth networks without issue. Reticulum automatically manages the flow of information to and from various network segments, and when bandwidth is limited, local traffic is prioritised.
- Reticulum provides sender/initiator anonymity by default. There is no way to filter traffic or discriminate it based on the source of the traffic.
- All traffic is encrypted using ephemeral keys generated by an Elliptic Curve Diffie-Hellman key exchange on Curve25519. There is no way to inspect traffic contents, and no way to prioritise or throttle certain kinds of traffic. All transport and routing layers are thus completely agnostic to traffic type, and will pass all traffic equally.
- Reticulum can function both with and without infrastructure. When *transport nodes* are available, they can route traffic over multiple hops for other nodes, and will function as a distributed cryptographic keystore. When there is no transport nodes available, all nodes that are within communication range can still communicate.
- Every node can become a transport node, simply by enabling it in its configuration, but there is no need for every node on the network to be a transport node. Letting every node be a transport node will in most cases degrade the performance and reliability of the network.

In general terms, if a node is stationary, well-connected and kept running most of the time, it is a good candidate to be a transport node. For optimal performance, a network should contain the amount of transport nodes that provides connectivity to the intended area / topography, and not many more than that.

- Reticulum is designed to work reliably in open, trustless environments. This means you can use it to create open-access networks, where participants can join and leave in a free and unorganised manner. This property

allows an entirely new, and so far, mostly unexplored class of networked applications, where networks, and the information flow within them can form and dissolve organically.

- You can just as easily create closed networks, since Reticulum allows you to add authentication to any interface. This means you can restrict access on any interface type, even when using legacy devices, such as modems. You can also mix authenticated and open interfaces on the same system. See the *Common Interface Options* section of the *Interfaces* chapter of this manual for information on how to set up interface authentication.

Reticulum allows you to mix very different kinds of networking mediums into a unified mesh, or to keep everything within one medium. You could build a “virtual network” running entirely over the Internet, where all nodes communicate over TCP and UDP “channels”. You could also build such a network using other already-established communications channels as the underlying carrier for Reticulum.

However, most real-world networks will probably involve either some form of wireless or direct hardline communications. To allow Reticulum to communicate over any type of medium, you must specify it in the configuration file, by default located at `~/.reticulum/config`. See the *Supported Interfaces* chapter of this manual for interface configuration examples.

Any number of interfaces can be configured, and Reticulum will automatically decide which are suitable to use in any given situation, depending on where traffic needs to flow.

4.2 Example Scenarios

This section illustrates a few example scenarios, and how they would, in general terms, be planned, implemented and configured.

4.2.1 Interconnected LoRa Sites

An organisation wants to provide communication and information services to its members, which are located mainly in three separate areas. Three suitable hill-top locations are found, where the organisation can install equipment: Site A, B and C.

Since the amount of data that needs to be exchanged between users is mainly text-based, the bandwidth requirements are low, and LoRa radios are chosen to connect users to the network.

Due to the hill-top locations found, there is radio line-of-sight between site A and B, and also between site B and C. Because of this, the organisation does not need to use the Internet to interconnect the sites, but purchases four Point-to-Point WiFi based radios for interconnecting the sites.

At each site, a Raspberry Pi is installed to function as a gateway. A LoRa radio is connected to the Pi with a USB cable, and the WiFi radio is connected to the ethernet port of the Pi. At site B, two WiFi radios are needed to be able to reach both site A and site C, so an extra ethernet adapter is connected to the Pi in this location.

Once the hardware has been installed, Reticulum is installed on all the Pis, and at site A and C, one interface is added for the LoRa radio, as well as one for the WiFi radio. At site B, an interface for the LoRa radio, and one interface for each WiFi radio is added to the Reticulum configuration file. The transport node option is enabled in the configuration of all three gateways.

The network is now operational, and ready to serve users across all three areas. The organisation prepares a LoRa radio that is supplied to the end users, along with a Reticulum configuration file, that contains the right parameters for communicating with the LoRa radios installed at the gateway sites.

Once users connect to the network, anyone will be able to communicate with anyone else across all three sites.

4.2.2 Bridging Over the Internet

As the organisation grows, several new communities form in places too far away from the core network to be reachable over WiFi links. New gateways similar to those previously installed are set up for the new communities at the new sites D and E, but they are islanded from the core network, and only serve the local users.

After investigating the options, it is found that it is possible to install an Internet connection at site A, and an interface on the Internet connection is configured for Reticulum on the Raspberry Pi at site A.

A member of the organisation at site D, named Dori, is willing to help by sharing the Internet connection she already has in her home, and is able to leave a Raspberry Pi running. A new Reticulum interface is configured on her Pi, connecting to the newly enabled Internet interface on the gateway at site A. Dori is now connected to both all the nodes at her own local site (through the hill-top LoRa gateway), and all the combined users of sites A, B and C. She then enables transport on her node, and traffic from site D can now reach everyone at site A, B and C, and vice versa.

4.2.3 Growth and Convergence

As the organisation grows, more gateways are added to keep up with the growing user base. Some local gateways even add VHF radios and packet modems to reach outlying users and communities that are out of reach for the LoRa radios and WiFi backhauls.

As more sites, gateways and users are connected, the amount of coordination required is kept to a minimum. If one community wants to add connectivity to the next one over, it can simply be done without having to involve everyone or coordinate address space or routing tables.

With the added geographical coverage, the operators at site A one day find that the original internet bridged interfaces are no longer utilised. The network has converged to be completely self-connected, and the sites that were once poorly connected outliers are now an integral part of the network.

SUPPORTED INTERFACES

Reticulum supports using many kinds of devices as networking interfaces, and allows you to mix and match them in any way you choose. The number of distinct network topologies you can create with Reticulum is more or less endless, but common to them all is that you will need to define one or more *interfaces* for Reticulum to use.

The following sections describe the interfaces currently available in Reticulum, and gives example configurations for the respective interface types.

For a high-level overview of how networks can be formed over different interface types, have a look at the *Building Networks* chapter of this manual.

5.1 Common Interface Options

A number of general configuration options are available on most interfaces. These can be used to control various aspects of interface behaviour.

- The `enabled` option tells Reticulum whether or not to bring up the interface. Defaults to `False`. For any interface to be brought up, the `enabled` option must be set to `True` or `Yes`.
- The `mode` option allows selecting the high-level behaviour of the interface from a number of options.
 - The default value is `full`. In this mode, all discovery, meshing and transport functionality is available.
 - In the `access_point` (or shorthand `ap`) mode, the interface will operate as a network access point. In this mode, announces will not be automatically broadcasted on the interface, and paths to destinations on the interface will have a much shorter expiry time. This mode is useful for creating interfaces that are mostly quiet, unless when someone is actually using them. An example of this could be a radio interface serving a wide area, where users are expected to connect momentarily, use the network, and then disappear again.
- The `outgoing` option sets whether an interface is allowed to transmit. Defaults to `True`. If set to `False` or `No` the interface will only receive data, and never transmit.
- The `network_name` option sets the virtual network name for the interface. This allows multiple separate network segments to exist on the same physical channel or medium.
- The `passphrase` option sets an authentication passphrase on the interface. This option can be used in conjunction with the `network_name` option, or be used alone.
- The `ifac_size` option allows customising the length of the Interface Authentication Codes carried by each packet on named and/or authenticated network segments. It is set by default to a size suitable for the interface in question, but can be set to a custom size between 8 and 512 bits by using this option. In normal usage, this option should not be changed from the default.
- The `announce_cap` option lets you configure the maximum bandwidth to allocate, at any given time, to propagating announces and other network upkeep traffic. It is configured at 2% by default, and should normally not need to be changed. Can be set to any value between 1 and 100.

- The `bitrate` option configures the interface bitrate. Reticulum will use interface speeds reported by hardware, or try to guess a suitable rate when the hardware doesn't report any. In most cases, the automatically found rate should be sufficient, but it can be configured by using the `bitrate` option, to set the interface speed in *bits per second*.

5.2 Auto Interface

The Auto Interface enables communication with other discoverable Reticulum nodes over autoconfigured IPv6 and UDP. It does not need any functional IP infrastructure like routers or DHCP servers, but will require at least some sort of switching medium between peers (a wired switch, a hub, a WiFi access point or similar), and that link-local IPv6 is enabled in your operating system, which should be enabled by default in almost all OSes.

```
# This example demonstrates a TCP server interface.
# It will listen for incoming connections on the
# specified IP address and port number.

[[Default Interface]]
  type = AutoInterface
  interface_enabled = True

  # You can create multiple isolated Reticulum
  # networks on the same physical LAN by
  # specifying different Group IDs.

  group_id = reticulum

  # You can also select specifically which
  # kernel networking devices to use.

  devices = wlan0,eth1

  # Or let AutoInterface use all suitable
  # devices except for a list of ignored ones.

  ignored_devices = tun0,eth0
```

If you are connected to the Internet with IPv6, and your provider will route IPv6 multicast, you can potentially configure the Auto Interface to globally autodiscover other Reticulum nodes within your selected Group ID. You can specify the discovery scope by setting it to one of `link`, `admin`, `site`, `organisation` or `global`.

```
[[Default Interface]]
  type = AutoInterface
  interface_enabled = True

  # Configure global discovery

  group_id = custom_network_name
  discovery_scope = global

  # Other configuration options

  discovery_port = 48555
```

(continues on next page)

(continued from previous page)

```
data_port = 49555
```

5.3 I2P Interface

The I2P interface lets you connect Reticulum instances over the [Invisible Internet Protocol](#). This can be especially useful in cases where you want to host a globally reachable Reticulum instance, but do not have access to any public IP addresses, have a frequently changing IP address, or have firewalls blocking inbound traffic.

Using the I2P interface, you will get a globally reachable, portable and persistent I2P address that your Reticulum instance can be reached at.

To use the I2P interface, you must have an I2P router running on your system. The easiest way to achieve this is to download and install the [latest release](#) of the `i2pd` package. For more details about I2P, see the [geti2p.net website](#).

When an I2P router is running on your system, you can simply add an I2P interface to reticulum:

```
[[I2P]]
type = I2PInterface
interface_enabled = yes
connectable = yes
```

On the first start, Reticulum will generate a new I2P address for the interface and start listening for inbound traffic on it. This can take a while the first time, especially if your I2P router was also just started, and is not yet well-connected to the I2P network. When ready, you should see I2P base32 address printed to your log file. You can also inspect the status of the interface using the `rnstatus` utility.

To connect to other Reticulum instances over I2P, just add a comma-separated list of I2P base32 addresses to the `peers` option of the interface:

```
[[I2P]]
type = I2PInterface
interface_enabled = yes
connectable = yes
peers = 5urvjicpzi7q3ybzts4i5ow2aq4soktfj7zedz53s47r54jnqq.b32.i2p
```

It can take anywhere from a few seconds to a few minutes to establish I2P connections to the desired peers, so Reticulum handles the process in the background, and will output relevant events to the log.

Please Note! While the I2P interface is the simplest way to use Reticulum over I2P, it is also possible to tunnel the TCP server and client interfaces over I2P manually. This can be useful in situations where more control is needed, but requires manual tunnel setup through the I2P daemon configuration.

It is important to note that the two methods are *interchangably compatible*. You can use the `I2PInterface` to connect to a `TCPServerInterface` that was manually tunneled over I2P, for example. This offers a high degree of flexibility in network setup, while retaining ease of use in simpler use-cases.

5.4 TCP Server Interface

The TCP Server interface is suitable for allowing other peers to connect over the Internet or private IP networks. When a TCP server interface has been configured, other Reticulum peers can connect to it with a TCP Client interface.

```
# This example demonstrates a TCP server interface.  
# It will listen for incoming connections on the  
# specified IP address and port number.
```

```
[[TCP Server Interface]]  
  type = TCPServerInterface  
  interface_enabled = True  
  
# This configuration will listen on all IP  
# interfaces on port 4242  
  
listen_ip = 0.0.0.0  
listen_port = 4242  
  
# Alternatively you can bind to a specific IP  
  
# listen_ip = 10.0.0.88  
# listen_port = 4242  
  
# Or a specific network device  
  
# device = eth0  
# port = 4242
```

Please Note! The TCP interfaces support tunneling over I2P, but to do so reliably, you must use the `i2p_tunneled` option:

```
[[TCP Server on I2P]]  
  type = TCPServerInterface  
  interface_enabled = yes  
  listen_ip = 127.0.0.1  
  listen_port = 5001  
  i2p_tunneled = yes
```

5.5 TCP Client Interface

To connect to a TCP server interface, you would naturally use the TCP client interface. Many TCP Client interfaces from different peers can connect to the same TCP Server interface at the same time.

```
# Here's an example of a TCP Client interface. The  
# target_host can either be an IP address or a hostname.
```

```
[[TCP Client Interface]]  
  type = TCPClientInterface  
  interface_enabled = True
```

(continues on next page)

(continued from previous page)

```
target_host = 127.0.0.1
target_port = 4242
```

It is also possible to use this interface type to connect via other programs or hardware devices that expose a KISS interface on a TCP port, for example software-based soundmodems. To do this, use the `kiss_framing` option:

```
# Here's an example of a TCP Client interface that connects
# to a software TNC soundmodem on a KISS over TCP port.
```

```
[[TCP KISS Interface]]
  type = TCPClientInterface
  interface_enabled = True
  kiss_framing = True
  target_host = 127.0.0.1
  target_port = 8001
```

Caution! Only use the KISS framing option when connecting to external devices and programs like soundmodems and similar over TCP. When using the `TCPClientInterface` in conjunction with the `TCPServerInterface` you should never enable `kiss_framing`, since this will disable internal reliability and recovery mechanisms that greatly improves performance over unreliable and intermittent TCP links.

Please Note! The TCP interfaces support tunneling over I2P, but to do so reliably, you must use the `i2p_tunneled` option:

```
[[TCP Client over I2P]]
  type = TCPClientInterface
  interface_enabled = yes
  target_host = 127.0.0.1
  target_port = 5001
  i2p_tunneled = yes
```

5.6 UDP Interface

A UDP interface can be useful for communicating over IP networks, both private and the internet. It can also allow broadcast communication over IP networks, so it can provide an easy way to enable connectivity with all other peers on a local area network.

Please Note! Using broadcast UDP traffic has performance implications, especially on WiFi. If your goal is simply to enable easy communication with all peers in your local ethernet broadcast domain, the *Auto Interface* performs better, and is just as easy to use.

The below example is enabled by default on new Reticulum installations, as it provides an easy way to get started and to test Reticulum on a pre-existing LAN.

```
# This example enables communication with other
# local Reticulum peers over UDP.
```

```
[[Default UDP Interface]]
  type = UDPInterface
  interface_enabled = True

  listen_ip = 0.0.0.0
```

(continues on next page)

(continued from previous page)

```
listen_port = 4242
forward_ip = 255.255.255.255
forward_port = 4242

# The above configuration will allow communication
# within the local broadcast domains of all local
# IP interfaces.

# Instead of specifying listen_ip, listen_port,
# forward_ip and forward_port, you can also bind
# to a specific network device like below.

# device = eth0
# port = 4242

# Assuming the eth0 device has the address
# 10.55.0.72/24, the above configuration would
# be equivalent to the following manual setup.
# Note that we are both listening and forwarding to
# the broadcast address of the network segments.

# listen_ip = 10.55.0.255
# listen_port = 4242
# forward_ip = 10.55.0.255
# forward_port = 4242

# You can of course also communicate only with
# a single IP address

# listen_ip = 10.55.0.15
# listen_port = 4242
# forward_ip = 10.55.0.16
# forward_port = 4242
```

5.7 RNode LoRa Interface

To use Reticulum over LoRa, the RNode interface can be used, and offers full control over LoRa parameters.

```
# Here's an example of how to add a LoRa interface
# using the RNode LoRa transceiver.

[[RNode LoRa Interface]]
  type = RNodeInterface

# Enable interface if you want use it!
interface_enabled = True

# Serial port for the device
port = /dev/ttyUSB0
```

(continues on next page)

(continued from previous page)

```

# Set frequency to 867.2 MHz
frequency = 867200000

# Set LoRa bandwidth to 125 KHz
bandwidth = 125000

# Set TX power to 7 dBm (5 mW)
txpower = 7

# Select spreading factor 8. Valid
# range is 7 through 12, with 7
# being the fastest and 12 having
# the longest range.
spreadingfactor = 8

# Select coding rate 5. Valid range
# is 5 through 8, with 5 being the
# fastest, and 8 the longest range.
codingrate = 5

# You can configure the RNode to send
# out identification on the channel with
# a set interval by configuring the
# following two parameters.
# id_callsign = MYCALL-0
# id_interval = 600

# For certain homebrew RNode interfaces
# with low amounts of RAM, using packet
# flow control can be useful. By default
# it is disabled.
flow_control = False

```

5.8 Serial Interface

Reticulum can be used over serial ports directly, or over any device with a serial port, that will transparently pass data. Useful for communicating directly over a wire-pair, or for using devices such as data radios and lasers.

```

[[Serial Interface]]
type = SerialInterface
interface_enabled = True

# Serial port for the device
port = /dev/ttyUSB0

# Set the serial baud-rate and other
# configuration parameters.
speed = 115200
databits = 8
parity = none

```

(continues on next page)

```
stopbits = 1
```

5.9 KISS Interface

With the KISS interface, you can use Reticulum over a variety of packet radio modems and TNCs, including [OpenModem](#). KISS interfaces can also be configured to periodically send out beacons for station identification purposes.

```
[[Packet Radio KISS Interface]]
  type = KISSInterface
  interface_enabled = True

  # Serial port for the device
  port = /dev/ttyUSB1

  # Set the serial baud-rate and other
  # configuration parameters.
  speed = 115200
  databits = 8
  parity = none
  stopbits = 1

  # Set the modem preamble.
  preamble = 150

  # Set the modem TX tail.
  txtail = 10

  # Configure CDMA parameters. These
  # settings are reasonable defaults.
  persistence = 200
  slottime = 20

  # You can configure the interface to send
  # out identification on the channel with
  # a set interval by configuring the
  # following two parameters. The KISS
  # interface will only ID if the set
  # interval has elapsed since it's last
  # actual transmission. The interval is
  # configured in seconds.
  # This option is commented out and not
  # used by default.
  # id_callsign = MYCALL-0
  # id_interval = 600

  # Whether to use KISS flow-control.
  # This is useful for modems that have
  # a small internal packet buffer, but
  # support packet flow control instead.
  flow_control = false
```

5.10 AX.25 KISS Interface

If you're using Reticulum on amateur radio spectrum, you might want to use the AX.25 KISS interface. This way, Reticulum will automatically encapsulate it's traffic in AX.25 and also identify your stations transmissions with your callsign and SSID.

Only do this if you really need to! Reticulum doesn't need the AX.25 layer for anything, and it incurs extra overhead on every packet to encapsulate in AX.25.

A more efficient way is to use the plain KISS interface with the beaconing functionality described above.

```
[[Packet Radio AX.25 KISS Interface]]
type = AX25KISSInterface

# Set the station callsign and SSID
callsign = N01CLL
ssid = 0

# Enable interface if you want use it!
interface_enabled = True

# Serial port for the device
port = /dev/ttyUSB2

# Set the serial baud-rate and other
# configuration parameters.
speed = 115200
databits = 8
parity = none
stopbits = 1

# Set the modem preamble. A 150ms
# preamble should be a reasonable
# default, but may need to be
# increased for radios with slow-
# opening squelch and long TX/RX
# turnaround
preamble = 150

# Set the modem TX tail. In most
# cases this should be kept as low
# as possible to not waste airtime.
txtail = 10

# Configure CDMA parameters. These
# settings are reasonable defaults.
persistence = 200
slottime = 20

# Whether to use KISS flow-control.
# This is useful for modems with a
# small internal packet buffer.
flow_control = false
```

UNDERSTANDING RETICULUM

This chapter will briefly describe the overall purpose and operating principles of Reticulum. It should give you an overview of how the stack works, and an understanding of how to develop networked applications using Reticulum.

This chapter is not an exhaustive source of information on Reticulum, at least not yet. Currently, the only complete repository, and final authority on how Reticulum actually functions, is the Python reference implementation and API reference. That being said, this chapter is an essential resource in understanding how Reticulum works from a high-level perspective, along with the general principles of Reticulum, and how to apply them when creating your own networks or software.

After reading this document, you should be well-equipped to understand how a Reticulum network operates, what it can achieve, and how you can use it yourself. If you want to help out with the development, this is also the place to start, since it will provide a pretty clear overview of the sentiments and the philosophy behind Reticulum, what problems it seeks to solve, and how it approaches those solutions.

6.1 Motivation

The primary motivation for designing and implementing Reticulum has been the current lack of reliable, functional and secure minimal-infrastructure modes of digital communication. It is my belief that it is highly desirable to create a reliable and efficient way to set up long-range digital communication networks that can securely allow exchange of information between people and machines, with no central point of authority, control, censorship or barrier to entry.

Almost all of the various networking systems in use today share a common limitation: They require large amounts of coordination and centralised trust and power to function. To join such networks, you need approval of gatekeepers in control. This need for coordination and trust inevitably leads to an environment of central control, where it's very easy for infrastructure operators or governments to control or alter traffic, and censor or persecute unwanted actors. It also makes it completely impossible to freely deploy and use networks at will, like one would use other common tools that enhance individual agency and freedom.

Reticulum aims to require as little coordination and trust as possible. It aims to make secure, anonymous and permissionless networking and information exchange a tool that anyone can just pick up and use.

Since Reticulum is completely medium agnostic, it can be used to build networks on whatever is best suited to the situation, or whatever you have available. In some cases, this might be packet radio links over VHF frequencies, in other cases it might be a 2.4 GHz network using off-the-shelf radios, or it might be using common LoRa development boards.

At the time of release of this document, the fastest and easiest setup for development and testing is using LoRa radio modules with an open source firmware (see the section *Reference Setup*), connected to any kind of computer or mobile device that Reticulum can run on.

The ultimate aim of Reticulum is to allow anyone to be their own network operator, and to make it cheap and easy to cover vast areas with a myriad of independent, interconnectable and autonomous networks. Reticulum **is not one network**, it **is a tool** to build *thousands of networks*.

Networks without kill-switches, surveillance, censorship and control. Networks that can freely interoperate, associate and disassociate with each other, and require no central oversight. Networks for human beings. *Networks for the people.*

6.2 Goals

To be as widely usable and efficient to deploy as possible, the following goals have been used to guide the design of Reticulum:

- **Fully useable as open source software stack** Reticulum must be implemented with, and be able to run using only open source software. This is critical to ensuring the availability, security and transparency of the system.
- **Hardware layer agnosticism** Reticulum must be fully hardware agnostic, and shall be useable over a wide range of physical networking layers, such as data radios, serial lines, modems, handheld transceivers, wired ethernet, wifi, or anything else that can carry a digital data stream. Hardware made for dedicated Reticulum use shall be as cheap as possible and use off-the-shelf components, so it can be easily modified and replicated by anyone interested in doing so.
- **Very low bandwidth requirements** Reticulum should be able to function reliably over links with a transmission capacity as low as *500 bits per second*.
- **Encryption by default** Reticulum must use strong encryption by default for all communication.
- **Initiator Anonymity** It must be possible to communicate over a Reticulum network without revealing any identifying information about oneself.
- **Unlicensed use** Reticulum shall be functional over physical communication mediums that do not require any form of license to use. Reticulum must be designed in a way, so it is usable over ISM radio frequency bands, and can provide functional long distance links in such conditions, for example by connecting a modem to a PMR or CB radio, or by using LoRa or WiFi modules.
- **Supplied software** In addition to the core networking stack and API, that allows a developer to build applications with Reticulum, a basic set of Reticulum-based communication tools must be implemented and released along with Reticulum itself. These shall serve both as a functional, basic communication suite, and as an example and learning resource to others wishing to build applications with Reticulum.
- **Ease of use** The reference implementation of Reticulum is written in Python, to make it easy to use and understand. A programmer with only basic experience should be able to use Reticulum to write networked applications.
- **Low cost** It shall be as cheap as possible to deploy a communication system based on Reticulum. This should be achieved by using cheap off-the-shelf hardware that potential users might already own. The cost of setting up a functioning node should be less than \$100 even if all parts needs to be purchased.

6.3 Introduction & Basic Functionality

Reticulum is a networking stack suited for high-latency, low-bandwidth links. Reticulum is at it's core a *message oriented* system. It is suited for both local point-to-point or point-to-multipoint scenarios where alle nodes are within range of each other, as well as scenarios where packets need to be transported over multiple hops in a complex network to reach the recipient.

Reticulum does away with the idea of addresses and ports known from IP, TCP and UDP. Instead Reticulum uses the singular concept of *destinations*. Any application using Reticulum as it's networking stack will need to create one or more destinations to receive data, and know the destinations it needs to send data to.

All destinations in Reticulum are represented as a 10 byte hash, derived from truncating a full SHA-256 hash of identifying characteristics of the destination. To users, the destination addresses will be displayed as 10 bytes in hexadecimal representation, as in the following example: <80e29bf7cccaf31431b3>.

The truncation size of 10 bytes (80 bits) for destinations has been chosen as a reasonable tradeoff between address space and packet overhead. The address space accommodated by this size can support many billions of simultaneously active devices on the same network, while keeping packet overhead low, which is essential on low-bandwidth networks. In the very unlikely case that this address space nears congestion, a one-line code change can upgrade the Reticulum address space all the way up to 256 bits, ensuring the Reticulum address space could potentially support galactic-scale networks. This is obviously complete and ridiculous over-allocation, and as such, the current 80 bits should be sufficient, even far into the future.

By default Reticulum encrypts all data using elliptic curve cryptography. Any packet sent to a destination is encrypted with a derived ephemeral key. Reticulum can also set up an encrypted channel to a destination with *Forward Secrecy* and *Initiator Anonymity* using a elliptic curve cryptography and ephemeral keys derived from a Diffie Hellman exchange on Curve25519. In Reticulum terminology, this is called a *Link*. The multi-hop transport, coordination, verification and reliability layers are fully autonomous and also based on elliptic curve cryptography.

Reticulum also offers symmetric key encryption for group-oriented communications, as well as unencrypted packets for broadcast purposes, or situations where you need the communication to be in plain text.

Reticulum can connect to a variety of interfaces such as radio modems, data radios and serial ports, and offers the possibility to easily tunnel Reticulum traffic over IP links such as the Internet or private IP networks.

6.3.1 Destinations

To receive and send data with the Reticulum stack, an application needs to create one or more destinations. Reticulum uses three different basic destination types, and one special:

- **Single** The *single* destination type is the most common type in Reticulum, and should be used for most purposes. It is always identified by a unique public key. Any data sent to this destination will be encrypted using ephemeral keys derived from an ECDH key exchange, and will only be readable by the creator of the destination, who holds the corresponding private key.
- **Plain** A *plain* destination type is unencrypted, and suited for traffic that should be broadcast to a number of users, or should be readable by anyone. Traffic to a *plain* destination is not encrypted. Generally, *plain* destinations can be used for broadcast information intended to be public. Plain destinations are only reachable directly, and packets addressed to plain destinations are never transported over multiple hops in the network. To be transportable over multiple hops in Reticulum, information *must* be encrypted, since Reticulum uses the per-packet encryption to verify routing paths and keep them alive.
- **Group** The *group* special destination type, that defines a symmetrically encrypted virtual destination. Data sent to this destination will be encrypted with a symmetric key, and will be readable by anyone in possession of the key, but as with the *plain* destination type, packets to this type of destination are not currently transported over multiple hops, although a planned upgrade to Reticulum will allow globally reachable *group* destinations.
- **Link** A *link* is a special destination type, that serves as an abstract channel to a *single* destination, directly connected or over multiple hops. The *link* also offers reliability and more efficient encryption, forward secrecy, initiator anonymity, and as such can be useful even when a node is directly reachable. It also offers a more capable API and allows easily carrying out requests and responses, large data transfers and more.

Destination Naming

Destinations are created and named in an easy to understand dotted notation of *aspects*, and represented on the network as a hash of this value. The hash is a SHA-256 truncated to 80 bits. The top level aspect should always be a unique identifier for the application using the destination. The next levels of aspects can be defined in any way by the creator of the application.

Aspects can be as long and as plentiful as required, and a resulting long destination name will not impact efficiency, as names are always represented as truncated SHA-256 hashes on the network.

As an example, a destination for an environmental monitoring application could be made up of the application name, a device type and measurement type, like this:

```
app name : environmentlogger
aspects  : remotesensor, temperature

full name : environmentlogger.remotesensor.temperature
hash      : fa7ddf5213f916dea
```

For the *single* destination, Reticulum will automatically append the associated public key as a destination aspect before hashing. This is done to ensure only the correct destination is reached, since anyone can listen to any destination name. Appending the public key ensures that a given packet is only directed at the destination that holds the corresponding private key to decrypt the packet.

Take note! There is a very important concept to understand here:

- Anyone can use the destination name `environmentlogger.remotesensor.temperature`
- Each destination that does so will still have a unique destination hash, and thus be uniquely addressable, because their public keys will differ.

In actual use of *single* destination naming, it is advisable not to use any uniquely identifying features in aspect naming. Aspect names should be general terms describing what kind of destination is represented. The uniquely identifying aspect is always achieved by the appending of the public key, which expands the destination into a uniquely identifiable one. Reticulum does this automatically.

Any destination on a Reticulum network can be addressed and reached simply by knowing its destination hash (and public key, but if the public key is not known, it can be requested from the network simply by knowing the destination hash). The use of app names and aspects makes it easy to structure Reticulum programs and makes it possible to filter what information and data your program receives.

To recap, the different destination types should be used in the following situations:

- **Single** When private communication between two endpoints is needed. Supports multiple hops.
- **Group** When private communication between two or more endpoints is needed. Supports multiple hops indirectly, but must first be established through a *single* destination.
- **Plain** When plain-text communication is desirable, for example when broadcasting information, or for local discovery purposes.

To communicate with a *single* destination, you need to know its public key. Any method for obtaining the public key is valid, but Reticulum includes a simple mechanism for making other nodes aware of your destination's public key, called the *announce*. It is also possible to request an unknown public key from the network, as all transport instances serve as a distributed ledger of public keys.

Note that public key information can be shared and verified in other ways than using the built-in *announce* functionality, and that it is therefore not required to use the *announce* and *path request* functionality to obtain public keys. It is by far the easiest though, and should definitely be used if there is not a very good reason for doing it differently.

6.3.2 Public Key Announcements

An *announce* will send a special packet over any relevant interfaces, containing all needed information about the destination hash and public key, and can also contain some additional, application specific data. The entire packet is signed by the sender to ensure authenticity. It is not required to use the announce functionality, but in many cases it will be the simplest way to share public keys on the network. The announce mechanism also serves to establish end-to-end connectivity to the announced destination, as the announce propagates through the network.

As an example, an announce in a simple messenger application might contain the following information:

- The announcers destination hash
- The announcers public key
- Application specific data, in this case the users nickname and availability status
- A random blob, making each new announce unique
- An Ed25519 signature of the above information, verifying authenticity

With this information, any Reticulum node that receives it will be able to reconstruct an outgoing destination to securely communicate with that destination. You might have noticed that there is one piece of information lacking to reconstruct full knowledge of the announced destination, and that is the aspect names of the destination. These are intentionally left out to save bandwidth, since they will be implicit in almost all cases. The receiving application will already know them. If a destination name is not entirely implicit, information can be included in the application specific data part that will allow the receiver to infer the naming.

It is important to note that announces will be forwarded throughout the network according to a certain pattern. This will be detailed in the section *The Announce Mechanism in Detail*.

In Reticulum, destinations are allowed to move around the network at will. This is very different from protocols such as IP, where an address is always expected to stay within the network segment it was assigned in. This limitation does not exist in Reticulum, and any destination is *completely portable* over the entire topography of the network, and *can even be moved to other Reticulum networks* than the one it was created in, and still become reachable. To update it's reachability, a destination simply needs to send an announce on any networks it is part of. After a short while, it will be globally reachable in the network.

Seeing how *single* destinations are always tied to a private/public key pair leads us to the next topic.

6.3.3 Identities

In Reticulum, an *identity* does not necessarily represent a personal identity, but is an abstraction that can represent any kind of *verifiable entity*. This could very well be a person, but it could also be the control interface of a machine, a program, robot, computer, sensor or something else entirely. In general, any kind of agent that can act, or be acted upon, or store or manipulate information, can be represented as an identity. An *identity* can be used to create any number of destinations.

A *single* destination will always have an *identity* tied to it, but not *plain* or *group* destinations. Destinations and identities share a multilateral connection. You can create a destination, and if it is not connected to an identity upon creation, it will just create a new one to use automatically. This may be desirable in some situations, but often you will probably want to create the identity first, and then use it to create new destinations.

As an example, we could use an identity to represent the user of a messaging application. Destinations can then be created by this identity to allow communication to reach the user. In all cases it is of great importance to store the private keys associated with any Reticulum Identity securely and privately, since obtaining access to the identity keys equals obtaining access and controlling reachability to any destinations created by that identity.

6.3.4 Getting Further

The above functions and principles form the core of Reticulum, and would suffice to create functional networked applications in local clusters, for example over radio links where all interested nodes can directly hear each other. But to be truly useful, we need a way to direct traffic over multiple hops in the network.

In the following sections, two concepts that allow this will be introduced, *paths* and *links*.

6.4 Reticulum Transport

The methods of routing used in traditional networks are fundamentally incompatible with the physical medium types and circumstances that Reticulum was designed to handle. These mechanisms mostly assume trust at the physical layer, and often needs a lot more bandwidth than Reticulum can assume is available. Since Reticulum is designed to survive running over open radio spectrum, no such trust can be assumed, and bandwidth is often very limited.

To overcome such challenges, Reticulum's *Transport* system uses asymmetric elliptic curve cryptography to implement the concept of *paths* that allow discovery of how to get information closer to a certain destination. It is important to note that no single node in a Reticulum network knows the complete path to a destination. Every Transport node participating in a Reticulum network will only know the most direct way to get a packet one hop closer to its destination.

6.4.1 Node Types

Currently, Reticulum distinguishes between two types of network nodes. All nodes on a Reticulum network are *Reticulum Instances*, and some are also *Transport Nodes*. If a system running Reticulum is fixed in one place, and is intended to be kept available most of the time, it is a good contender to be a *Transport Node*.

Any Reticulum Instance can become a Transport Node by enabling it in the configuration. This distinction is made by the user configuring the node, and is used to determine what nodes on the network will help forward traffic, and what nodes rely on other nodes for wider connectivity.

If a node is an *Instance* it should be given the configuration directive `enable_transport = No`, which is the default setting.

If it is a *Transport Node*, it should be given the configuration directive `enable_transport = Yes`.

6.4.2 The Announce Mechanism in Detail

When an *announce* for a destination is transmitted by from a Reticulum instance, it will be forwarded by any transport node receiving it, but according to some specific rules:

- If this exact announce has already been received before, ignore it.
- If not, record into a table which Transport Node the announce was received from, and how many times in total it has been retransmitted to get here.
- If the announce has been retransmitted $m+1$ times, it will not be forwarded any more. By default, m is set to 128.
- After a randomised delay, the announce will be retransmitted on all interfaces that have bandwidth available for processing announces. By default, the maximum bandwidth allocation for processing announces is set at 2%, but can be configured on a per-interface basis.
- If any given interface does not have enough bandwidth available for retransmitting the announce, the announce will be assigned a priority inversely proportional to its hop count, and be inserted into a queue managed by the interface.

- When the interface has bandwidth available for processing an announce, it will prioritise announces for destinations that are closest in terms of hops, thus prioritising reachability and connectivity of local nodes, even on slow networks that connect to wider and faster networks.
- After the announce has been re-transmitted, and if no other nodes are heard retransmitting the announce with a greater hop count than when it left this node, transmitting it will be retried r times. By default, r is set to 1.
- If a newer announce from the same destination arrives, while an identical one is already waiting to be transmitted, the newest announce is discarded. If the newest announce contains different application specific data, it will replace the old announce.

Once an announce has reached a node in the network, any other node in direct contact with that node will be able to reach the destination the announce originated from, simply by sending a packet addressed to that destination. Any node with knowledge of the announce will be able to direct the packet towards the destination by looking up the next node with the shortest amount of hops to the destination.

According to these rules, an announce will propagate throughout the network in a predictable way, and make the announced destination reachable in a short amount of time. Fast networks that have the capacity to process many announces can reach full convergence very quickly, even when constantly adding new destinations. Slower segments of such networks might take a bit longer to gain full knowledge about the wide and fast networks they are connected to, but can still do so over time, while prioritising full and quickly converging end-to-end connectivity for their local, slower segments.

In general, even extremely complex networks, that utilize the maximum 128 hops will converge to full end-to-end connectivity in about one minute, given there is enough bandwidth available to process the required amount of announces.

6.4.3 Reaching the Destination

In networks with changing topology and trustless connectivity, nodes need a way to establish *verified connectivity* with each other. Since the network is assumed to be trustless, Reticulum must provide a way to guarantee that the peer you are communicating with is actually who you expect. Reticulum offers two ways to do this.

For exchanges of small amounts of information, Reticulum offers the *Packet API*, which works exactly like you would expect - on a per packet level. The following process is employed when sending a packet:

- A packet is always created with an associated destination and some payload data. When the packet is sent to a *single* destination type, Reticulum will automatically create an ephemeral encryption key, perform an ECDH key exchange with the destinations public key, and encrypt the information.
- It is important to note that this key exchange does not require any network traffic. The sender already knows the public key of the destination from an earlier received *announce*, and can thus perform the ECDH key exchange locally, before sending the packet.
- The public part of the newly generated ephemeral key-pair is included with the encrypted token, and sent along with the encrypted payload data in the packet.
- When the destination receives the packet, it can itself perform an ECDH key exchange and decrypt the packet.
- A new ephemeral key is used for every packet sent in this way, and forward secrecy is guaranteed on a per packet level.
- Once the packet has been received and decrypted by the addressed destination, that destination can opt to *prove* its receipt of the packet. It does this by calculating the SHA-256 hash of the received packet, and signing this hash with it's Ed25519 signing key. Transport nodes in the network can then direct this *proof* back to the packets origin, where the signature can be verified against the destinations known public signing key.
- In case the packet is addressed to a *group* destination type, the packet will be encrypted with the pre-shared AES-128 key associated with the destination. In case the packet is addressed to a *plain* destination type, the payload data will not be encrypted. Neither of these two destination types offer forward secrecy. In general, it is recommended to always use the *single* destination type, unless it is strictly necessary to use one of the others.

For exchanges of larger amounts of data, or when longer sessions of bidirectional communication is desired, Reticulum offers the *Link* API. To establish a *link*, the following process is employed:

- First, the node that wishes to establish a link will send out a special packet, that traverses the network and locates the desired destination. Along the way, the Transport Nodes that forward the packet will take note of this *link request*.
- Second, if the destination accepts the *link request*, it will send back a packet that proves the authenticity of its identity (and the receipt of the link request) to the initiating node. All nodes that initially forwarded the packet will also be able to verify this proof, and thus accept the validity of the *link* throughout the network.
- When the validity of the *link* has been accepted by forwarding nodes, these nodes will remember the *link*, and it can subsequently be used by referring to a hash representing it.
- As a part of the *link request*, an Elliptic Curve Diffie-Hellman key exchange takes place, that sets up an efficiently encrypted tunnel between the two nodes. As such, this mode of communication is preferred, even for situations when nodes can directly communicate, when the amount of data to be exchanged numbers in the tens of packets, or whenever the use of the more advanced API functions is desired.
- When a *link* has been set up, it automatically provides message receipt functionality, through the same *proof* mechanism discussed before, so the sending node can obtain verified confirmation that the information reached the intended recipient.
- Once the *link* has been set up, the initiator can remain anonymous, or choose to authenticate towards the destination using a Reticulum Identity. This authentication is happening inside the encrypted link, and is only revealed to the verified destination, and no intermediaries.

In a moment, we will discuss the details of how this methodology is implemented, but let's first recap what purposes this methodology serves. We first ensure that the node answering our request is actually the one we want to communicate with, and not a malicious actor pretending to be so. At the same time we establish an efficient encrypted channel. The setup of this is relatively cheap in terms of bandwidth, so it can be used just for a short exchange, and then recreated as needed, which will also rotate encryption keys. The link can also be kept alive for longer periods of time, if this is more suitable to the application. The procedure also inserts the *link id*, a hash calculated from the link request packet, into the memory of forwarding nodes, which means that the communicating nodes can thereafter reach each other simply by referring to this *link id*.

The combined bandwidth cost of setting up a link is 3 packets totalling 237 bytes (more info in the *Binary Packet Format* section). The amount of bandwidth used on keeping a link open is practically negligible, at 0.62 bits per second. Even on a slow 1200 bits per second packet radio channel, 100 concurrent links will still leave 95% channel capacity for actual data.

Link Establishment in Detail

After exploring the basics of the announce mechanism, finding a path through the network, and an overview of the link establishment procedure, this section will go into greater detail about the Reticulum link establishment process.

The *link* in Reticulum terminology should not be viewed as a direct node-to-node link on the physical layer, but as an abstract channel, that can be open for any amount of time, and can span an arbitrary number of hops, where information will be exchanged between two nodes.

- When a node in the network wants to establish verified connectivity with another node, it will randomly generate a new X25519 private/public key pair. It then creates a *link request* packet, and broadcast it.

It should be noted that the X25519 public/private keypair mentioned above is two separate keypairs: An encryption key pair, used for derivation of a shared symmetric key, and a signing key pair, used for signing and verifying messages on the link. They are sent together over the wire, and can be considered as single public key for simplicity in this explanation.

- The *link request* is addressed to the destination hash of the desired destination, and contains the following data: The newly generated X25519 public key *LKi*.
- The broadcasted packet will be directed through the network according to the rules laid out previously.
- Any node that forwards the link request will store a *link id* in its *link table*, along with the amount of hops the packet had taken when received. The link id is a hash of the entire link request packet. If the link request packet is not *proven* by the addressed destination within some set amount of time, the entry will be dropped from the *link table* again.
- When the destination receives the link request packet, it will decide whether to accept the request. If it is accepted, the destination will also generate a new X25519 private/public key pair, and perform a Diffie Hellman Key Exchange, deriving a new symmetric key that will be used to encrypt the channel, once it has been established.
- A *link proof* packet is now constructed and transmitted over the network. This packet is addressed to the *link id* of the *link*. It contains the following data: The newly generated X25519 public key *LKr* and an Ed25519 signature of the *link id* and *LKr* made by the signing key of the addressed destination.
- By verifying this *link proof* packet, all nodes that originally transported the *link request* packet to the destination from the originator can now verify that the intended destination received the request and accepted it, and that the path they chose for forwarding the request was valid. In successfully carrying out this verification, the transporting nodes marks the link as active. An abstract bi-directional communication channel has now been established along a path in the network.
- When the source receives the *proof*, it will know unequivocally that a verified path has been established to the destination. It can now also use the X25519 public key contained in the *link proof* to perform its own Diffie Hellman Key Exchange and derive the symmetric key that is used to encrypt the channel. Information can now be exchanged reliably and securely.

It's important to note that this methodology ensures that the source of the request does not need to reveal any identifying information about itself. The link initiator remains completely anonymous.

When using *links*, Reticulum will automatically verify all data sent over the link, and can also automate retransmissions if *Resources* are used.

6.4.4 Resources

For exchanging small amounts of data over a Reticulum network, the *Packet* interface is sufficient, but for exchanging data that would require many packets, an efficient way to coordinate the transfer is needed.

This is the purpose of the Reticulum *Resource*. A *Resource* can automatically handle the reliable transfer of an arbitrary amount of data over an established *Link*. Resources can auto-compress data, will handle breaking the data into individual packets, sequencing the transfer, integrity verification and reassembling the data on the other end.

Resources are programmatically very simple to use, and only requires a few lines of codes to reliably transfer any amount of data. They can be used to transfer data stored in memory, or stream data directly from files.

6.5 Reference Setup

This section will detail a recommended *Reference Setup* for Reticulum. It is important to note that Reticulum is designed to be usable on more or less any computing device, and over more or less any medium that allows you to send and receive data, which satisfies some very low minimum requirements.

The communication channel must support at least half-duplex operation, and provide an average throughput of around 500 bits per second, and supports a physical layer MTU of 500 bytes. The Reticulum stack should be able to run on more or less any hardware that can provide a Python 3.x runtime environment.

That being said, this reference setup has been outlined to provide a common platform for anyone who wants to help in the development of Reticulum, and for everyone who wants to know a recommended setup to get started experimenting. A reference system consists of three parts:

- **An Interface Device** Which provides access to the physical medium whereupon the communication takes place, for example a radio with an integrated modem. A setup with a separate modem connected to a radio would also be an interface device.
- **A Host Device** Some sort of computing device that can run the necessary software, communicate with the interface device, and provide user interaction.
- **A Software Stack** The software implementing the Reticulum protocol and applications using it.

The reference setup can be considered a relatively stable platform to develop on, and also to start building networks or applications on. While details of the implementation might change at the current stage of development, it is the goal to maintain hardware compatibility for as long as entirely possible, and the current reference setup has been determined to provide a functional platform for many years into the future. The current Reference System Setup is as follows:

- **Interface Device** A data radio consisting of a LoRa radio module, and a microcontroller with open source firmware, that can connect to host devices via USB. It operates in either the 430, 868 or 900 MHz frequency bands. More details can be found on the [RNode Page](#).
- **Host Device** Any computer device running Linux and Python. A Raspberry Pi with a Debian based OS is recommended.
- **Software Stack** The most recently released Python Implementation of Reticulum, running on a Debian based operating system.

To avoid confusion, it is very important to note, that the reference interface device **does not** use the LoRaWAN standard, but uses a custom MAC layer on top of the plain LoRa modulation! As such, you will need a plain LoRa radio module connected to an controller with the correct firmware. Full details on how to get or make such a device is available on the [RNode Page](#).

With the current reference setup, it should be possible to get on a Reticulum network for around 100\$ even if you have none of the hardware already, and need to purchase everything.

This reference setup is of course just a recommendation for getting started easily, and you should tailor it to your own specific needs, or whatever hardware you have available.

6.6 Protocol Specifics

This chapter will detail protocol specific information that is essential to the implementation of Reticulum, but non critical in understanding how the protocol works on a general level. It should be treated more as a reference than as essential reading.

6.6.1 Packet Prioritisation

Currently, Reticulum is completely priority-agnostic regarding general traffic. All traffic is handled on a first-come, first-serve basis. Announce re-transmission are handled according to the re-transmission times and priorities described earlier in this chapter.

6.6.2 Interface Access Codes

Reticulum can create named virtual networks, and networks that are only accessible by knowing a preshared passphrase. The configuration of this is detailed in the *Common Interface Options* section. To implement these feature, Reticulum uses the concept of Interface Access Codes, that are calculated and verified per packet.

An interface with a named virtual network or passphrase authentication enabled will derive a shared Ed25519 signing identity, and for every outbound packet generate a signature of the entire packet. This signature is then inserted into the packet as an Interface Access Code before transmission. Depending on the speed and capabilities of the interface, the IFAC can be the full 512-bit Ed25519 signature, or a truncated version. Configured IFAC length can be inspected for all interfaces with the `rnstatus` utility.

Upon receipt, the interface will check that the signature matches the expected value, and drop the packet if it does not. This ensures that only packets sent with the correct naming and/or passphrase parameters are allowed to pass onto the network.

6.6.3 Wire Format

```
== Reticulum Wire Format =====

A Reticulum packet is composed of the following fields:

[HEADER 2 bytes] [ADDRESSES 10/20 bytes] [CONTEXT 1 byte] [DATA 0-477 bytes]

* The HEADER field is 2 bytes long.
  * Byte 1: [IFAC Flag], [Header Type], [Propagation Type], [Destination Type] and
  ↪ [Packet Type]
  * Byte 2: Number of hops

* Interface Access Code field if the IFAC flag was set.
  * The length of the Interface Access Code can vary from
    1 to 64 bytes according to physical interface
    capabilities and configuration.

* The ADDRESSES field contains either 1 or 2 addresses.
  * Each address is 10 bytes long.
  * The Header Type flag in the HEADER field determines
    whether the ADDRESSES field contains 1 or 2 addresses.
  * Addresses are Reticulum hashes truncated to 10 bytes.
```

(continues on next page)

(continued from previous page)

* The CONTEXT field is 1 byte.
 * It is used by Reticulum to determine packet context.

* The DATA field is between 0 and 477 bytes.
 * It contains the packets data payload.

IFAC Flag

open 0 Packet for publically accessible interface
 authenticated 1 Interface authentication is included in packet

Header Types

type 1 0 Two byte header, one 10 byte address field
 type 2 1 Two byte header, two 10 byte address fields

Propagation Types

broadcast 00
 transport 01
 reserved 10
 reserved 11

Destination Types

single 00
 group 01
 plain 10
 link 11

Packet Types

data 00
 announce 01
 link request 10
 proof 11

+ - Packet Example - +

HEADER FIELD	DESTINATION FIELDS	CONTEXT FIELD	DATA FIELD
----- -----	----- -----	----- -----	-- -
01010000	00000100 [HASH1, 10 bytes] [HASH2, 10 bytes]	[CONTEXT, 1 byte]	[DATA]
	+-- Hops	= 4	
	+----- Packet Type	= DATA	

(continues on next page)

(continued from previous page)

```

|| | +----- Destination Type = SINGLE
|| +----- Propagation Type = TRANSPORT
|+----- Header Type = HEADER_2 (two byte header, two address fields)
+----- Access Codes = DISABLED

```

+ - Packet Example - +

```

      HEADER FIELD  DESTINATION FIELD  CONTEXT FIELD  DATA FIELD
-----|-----  -----|-----  -----|-----  ---|---
|         |         |         |         |         |         |         |
00000000 00000111 [HASH1, 10 bytes] [CONTEXT, 1 byte] [DATA]
|| | | |         |
|| | | |   +-- Hops           = 0
|| | | +----- Packet Type   = DATA
|| | +----- Destination Type = SINGLE
|| +----- Propagation Type = BROADCAST
|+----- Header Type = HEADER_1 (two byte header, one address field)
+----- Access Codes = DISABLED

```

+ - Packet Example - +

```

      HEADER FIELD  IFAC FIELD  DESTINATION FIELD  CONTEXT FIELD  DATA FIELD
-----|-----  -----|-----  -----|-----  -----|-----  ---|---
|         |         |         |         |         |         |         |
10000000 00000111 [IFAC, N bytes] [HASH1, 10 bytes] [CONTEXT, 1 byte] [DATA]
|| | | |         |
|| | | |   +-- Hops           = 0
|| | | +----- Packet Type   = DATA
|| | +----- Destination Type = SINGLE
|| +----- Propagation Type = BROADCAST
|+----- Header Type = HEADER_1 (two byte header, one address field)
+----- Access Codes = ENABLED

```

Size examples of different packet types

The following table lists example sizes of various packet types. The size listed are the complete on-wire size counting all fields including headers, but excluding any interface access codes.

```

- Path Request   : 33 bytes
- Announce       : 151 bytes
- Link Request   : 77 bytes
- Link Proof     : 77 bytes
- Link RTT packet : 83 bytes
- Link keepalive : 14 bytes

```


API REFERENCE

This reference guide lists and explains all classes exposed by the RNS API.

7.1 Classes

Communication over a Reticulum network is achieved using a set of classes exposed by RNS.

7.1.1 Reticulum

class `RNS.Reticulum`(*configdir=None, loglevel=None*)

This class is used to initialise access to Reticulum within a program. You must create exactly one instance of this class before carrying out any other RNS operations, such as creating destinations or sending traffic. Every independently executed program must create their own instance of the Reticulum class, but Reticulum will automatically handle inter-program communication on the same system, and expose all connected programs to external interfaces as well.

As soon as an instance of this class is created, Reticulum will start opening and configuring any hardware devices specified in the supplied configuration.

Currently the first running instance must be kept running while other local instances are connected, as the first created instance will act as a master instance that directly communicates with external hardware such as modems, TNCs and radios. If a master instance is asked to exit, it will not exit until all client processes have terminated (unless killed forcibly).

If you are running Reticulum on a system with several different programs that use RNS starting and terminating at different times, it will be advantageous to run a master RNS instance as a daemon for other programs to use on demand.

MTU = 500

The MTU that Reticulum adheres to, and will expect other peers to adhere to. By default, the MTU is 507 bytes. In custom RNS network implementations, it is possible to change this value, but doing so will completely break compatibility with all other RNS networks. An identical MTU is a prerequisite for peers to communicate in the same network.

Unless you really know what you are doing, the MTU should be left at the default value.

ANNOUNCE_CAP = 2

The maximum percentage of interface bandwidth that, at any given time, may be used to propagate announcements. If an announce was scheduled for broadcasting on an interface, but doing so would exceed the allowed bandwidth allocation, the announce will be queued for transmission when there is bandwidth available.

Reticulum will always prioritise propagating announces with fewer hops, ensuring that distant, large networks with many peers on fast links don't overwhelm the capacity of smaller networks on slower mediums. If an announce remains queued for an extended amount of time, it will eventually be dropped.

This value will be applied by default to all created interfaces, but it can be configured individually on a per-interface basis.

static should_use_implicit_proof()

Returns whether proofs sent are explicit or implicit.

Returns True if the current running configuration specifies to use implicit proofs. False if not.

static transport_enabled()

Returns whether Transport is enabled for the running instance.

When Transport is enabled, Reticulum will route traffic for other peers, respond to path requests and pass announces over the network.

Returns True if Transport is enabled, False if not.

7.1.2 Identity

class `RNS.Identity`(*create_keys=True*)

This class is used to manage identities in Reticulum. It provides methods for encryption, decryption, signatures and verification, and is the basis for all encrypted communication over Reticulum networks.

Parameters `create_keys` – Specifies whether new encryption and signing keys should be generated.

CURVE = 'Curve25519'

The curve used for Elliptic Curve DH key exchanges

KEYSIZE = 512

X25519 key size in bits. A complete key is the concatenation of a 256 bit encryption key, and a 256 bit signing key.

TRUNCATED_HASHLENGTH = 80

Constant specifying the truncated hash length (in bits) used by Reticulum for addressable hashes and other purposes. Non-configurable.

static recall(*destination_hash*)

Recall identity for a destination hash.

Parameters `destination_hash` – Destination hash as *bytes*.

Returns An `RNS.Identity` instance that can be used to create an outgoing `RNS.Destination`, or `None` if the destination is unknown.

static recall_app_data(*destination_hash*)

Recall last heard `app_data` for a destination hash.

Parameters `destination_hash` – Destination hash as *bytes*.

Returns `Bytes` containing `app_data`, or `None` if the destination is unknown.

static full_hash(*data*)

Get a SHA-256 hash of passed data.

Parameters `data` – Data to be hashed as *bytes*.

Returns SHA-256 hash as *bytes*

static truncated_hash(*data*)

Get a truncated SHA-256 hash of passed data.

Parameters **data** – Data to be hashed as *bytes*.

Returns Truncated SHA-256 hash as *bytes*

static get_random_hash()

Get a random SHA-256 hash.

Parameters **data** – Data to be hashed as *bytes*.

Returns Truncated SHA-256 hash of random data as *bytes*

static from_bytes(*prv_bytes*)

Create a new *RNS.Identity* instance from *bytes* of private key. Can be used to load previously created and saved identities into Reticulum.

Parameters **prv_bytes** – The *bytes* of private a saved private key. **HAZARD!** Never use this to generate a new key by feeding random data in *prv_bytes*.

Returns A *RNS.Identity* instance, or *None* if the *bytes* data was invalid.

static from_file(*path*)

Create a new *RNS.Identity* instance from a file. Can be used to load previously created and saved identities into Reticulum.

Parameters **path** – The full path to the saved *RNS.Identity* data

Returns A *RNS.Identity* instance, or *None* if the loaded data was invalid.

to_file(*path*)

Saves the identity to a file. This will write the private key to disk, and anyone with access to this file will be able to decrypt all communication for the identity. Be very careful with this method.

Parameters **path** – The full path specifying where to save the identity.

Returns True if the file was saved, otherwise False.

get_private_key()

Returns The private key as *bytes*

get_public_key()

Returns The public key as *bytes*

load_private_key(*prv_bytes*)

Load a private key into the instance.

Parameters **prv_bytes** – The private key as *bytes*.

Returns True if the key was loaded, otherwise False.

load_public_key(*pub_bytes*)

Load a public key into the instance.

Parameters **pub_bytes** – The public key as *bytes*.

Returns True if the key was loaded, otherwise False.

encrypt(*plaintext*)

Encrypts information for the identity.

Parameters **plaintext** – The plaintext to be encrypted as *bytes*.

Returns Ciphertext token as *bytes*.

Raises *KeyError* if the instance does not hold a public key.

decrypt(*ciphertext_token*)

Decrypts information for the identity.

Parameters **ciphertext** – The ciphertext to be decrypted as *bytes*.

Returns Plaintext as *bytes*, or *None* if decryption fails.

Raises *KeyError* if the instance does not hold a private key.

sign(*message*)

Signs information by the identity.

Parameters **message** – The message to be signed as *bytes*.

Returns Signature as *bytes*.

Raises *KeyError* if the instance does not hold a private key.

validate(*signature, message*)

Validates the signature of a signed message.

Parameters

- **signature** – The signature to be validated as *bytes*.
- **message** – The message to be validated as *bytes*.

Returns True if the signature is valid, otherwise False.

Raises *KeyError* if the instance does not hold a public key.

7.1.3 Destination

class **RNS.Destination**(*identity, direction, type, app_name, *aspects*)

A class used to describe endpoints in a Reticulum Network. Destination instances are used both to create outgoing and incoming endpoints. The destination type will decide if encryption, and what type, is used in communication with the endpoint. A destination can also announce its presence on the network, which will also distribute necessary keys for encrypted communication with it.

Parameters

- **identity** – An instance of *RNS.Identity*. Can hold only public keys for an outgoing destination, or holding private keys for an ingoing.
- **direction** – *RNS.Destination.IN* or *RNS.Destination.OUT*.
- **type** – *RNS.Destination.SINGLE*, *RNS.Destination.GROUP* or *RNS.Destination.PLAIN*.
- **app_name** – A string specifying the app name.
- ***aspects** – Any non-zero number of string arguments.

static **full_name**(*app_name, *aspects*)

Returns A string containing the full human-readable name of the destination, for an *app_name* and a number of aspects.

static **app_and_aspects_from_name**(*full_name*)

Returns A tuple containing the app name and a list of aspects, for a full-name string.

static hash_from_name_and_identity(*full_name, identity*)

Returns A destination name in adressable hash form, for a full name string and Identity instance.

static hash(*app_name, *aspects*)

Returns A destination name in adressable hash form, for an app_name and a number of aspects.

announce(*app_data=None, path_response=False*)

Creates an announce packet for this destination and broadcasts it on all relevant interfaces. Application specific data can be added to the announce.

Parameters

- **app_data** – *bytes* containing the app_data.
- **path_response** – Internal flag used by *RNS.Transport*. Ignore.

set_link_established_callback(*callback*)

Registers a function to be called when a link has been established to this destination.

Parameters **callback** – A function or method to be called.

set_packet_callback(*callback*)

Registers a function to be called when a packet has been received by this destination.

Parameters **callback** – A function or method to be called.

set_proof_requested_callback(*callback*)

Registers a function to be called when a proof has been requested for a packet sent to this destination. Allows control over when and if proofs should be returned for received packets.

Parameters **callback** – A function or method to be called. The callback must return one of True or False. If the callback returns True, a proof will be sent. If it returns False, a proof will not be sent.

set_proof_strategy(*proof_strategy*)

Sets the destinations proof strategy.

Parameters **proof_strategy** – One of *RNS.Destination.PROVE_NONE*, *RNS.Destination.PROVE_ALL* or *RNS.Destination.PROVE_APP*. If *RNS.Destination.PROVE_APP* is set, the *proof_requested_callback* will be called to determine whether a proof should be sent or not.

register_request_handler(*path, response_generator=None, allow=0, allowed_list=None*)

Registers a request handler.

Parameters

- **path** – The path for the request handler to be registered.
- **response_generator** – A function or method with the signature *response_generator(path, data, request_id, remote_identity, requested_at)* to be called. Whatever this function returns will be sent as a response to the requester. If the function returns None, no response will be sent.
- **allow** – One of *RNS.Destination.ALLOW_NONE*, *RNS.Destination.ALLOW_ALL* or *RNS.Destination.ALLOW_LIST*. If *RNS.Destination.ALLOW_LIST* is set, the request handler will only respond to requests for identified peers in the supplied list.
- **allowed_list** – A list of *bytes-like RNS.Identity* hashes.

Raises ValueError if any of the supplied arguments are invalid.

deregister_request_handler(*path*)

Deregisters a request handler.

Parameters *path* – The path for the request handler to be deregistered.

Returns True if the handler was deregistered, otherwise False.

create_keys()

For a RNS.Destination.GROUP type destination, creates a new symmetric key.

Raises TypeError if called on an incompatible type of destination.

get_private_key()

For a RNS.Destination.GROUP type destination, returns the symmetric private key.

Raises TypeError if called on an incompatible type of destination.

load_private_key(*key*)

For a RNS.Destination.GROUP type destination, loads a symmetric private key.

Parameters *key* – A *bytes-like* containing the symmetric key.

Raises TypeError if called on an incompatible type of destination.

encrypt(*plaintext*)

Encrypts information for RNS.Destination.SINGLE or RNS.Destination.GROUP type destination.

Parameters *plaintext* – A *bytes-like* containing the plaintext to be encrypted.

Raises ValueError if destination does not hold a necessary key for encryption.

decrypt(*ciphertext*)

Decrypts information for RNS.Destination.SINGLE or RNS.Destination.GROUP type destination.

Parameters *ciphertext* – *Bytes* containing the ciphertext to be decrypted.

Raises ValueError if destination does not hold a necessary key for decryption.

sign(*message*)

Signs information for RNS.Destination.SINGLE type destination.

Parameters *message* – *Bytes* containing the message to be signed.

Returns A *bytes-like* containing the message signature, or *None* if the destination could not sign the message.

set_default_app_data(*app_data=None*)

Sets the default *app_data* for the destination. If set, the default *app_data* will be included in every announce sent by the destination, unless other *app_data* is specified in the *announce* method.

Parameters *app_data* – A *bytes-like* containing the default *app_data*, or a *callable* returning a *bytes-like* containing the *app_data*.

clear_default_app_data()

Clears default *app_data* previously set for the destination.

7.1.4 Packet

class `RNS.Packet`(*destination*, *data*, *create_receipt=True*)

The Packet class is used to create packet instances that can be sent over a Reticulum network. Packets will automatically be encrypted if they are addressed to a `RNS.Destination.SINGLE` destination, `RNS.Destination.GROUP` destination or a `RNS.Link`.

For `RNS.Destination.GROUP` destinations, Reticulum will use the pre-shared key configured for the destination.

For `RNS.Destination.SINGLE` destinations and `RNS.Link` destinations, reticulum will use ephemeral keys, and offers **Forward Secrecy**.

Parameters

- **destination** – A `RNS.Destination` instance to which the packet will be sent.
- **data** – The data payload to be included in the packet as *bytes*.
- **create_receipt** – Specifies whether a `RNS.PacketReceipt` should be created when instantiating the packet.

ENCRYPTED_MDU = 383

The maximum size of the payload data in a single encrypted packet

PLAIN_MDU = 476

The maximum size of the payload data in a single unencrypted packet

send()

Sends the packet.

Returns A `RNS.PacketReceipt` instance if *create_receipt* was set to *True* when the packet was instantiated, if not returns *None*. If the packet could not be sent *False* is returned.

resend()

Re-sends the packet.

Returns A `RNS.PacketReceipt` instance if *create_receipt* was set to *True* when the packet was instantiated, if not returns *None*. If the packet could not be sent *False* is returned.

7.1.5 Packet Receipt

class `RNS.PacketReceipt`

The PacketReceipt class is used to receive notifications about `RNS.Packet` instances sent over the network. Instances of this class are never created manually, but always returned from the `send()` method of a `RNS.Packet` instance.

get_status()

Returns The status of the associated `RNS.Packet` instance. Can be one of `RNS.PacketReceipt.SENT`, `RNS.PacketReceipt.DELIVERED`, `RNS.PacketReceipt.FAILED` or `RNS.PacketReceipt.CULLED`.

get_rtt()

Returns The round-trip-time in seconds

set_timeout(*timeout*)

Sets a timeout in seconds

Parameters `timeout` – The timeout in seconds.

`set_delivery_callback(callback)`

Sets a function that gets called if a successful delivery has been proven.

Parameters `callback` – A callable with the signature `callback(packet_receipt)`

`set_timeout_callback(callback)`

Sets a function that gets called if the delivery times out.

Parameters `callback` – A callable with the signature `callback(packet_receipt)`

7.1.6 Link

class `RNS.Link(destination, established_callback=None, closed_callback=None)`

This class is used to establish and manage links to other peers. When a link instance is created, Reticulum will attempt to establish verified connectivity with the specified destination.

Parameters

- **destination** – A `RNS.Destination` instance which to establish a link to.
- **established_callback** – An optional function or method with the signature `callback(link)` to be called when the link has been established.
- **closed_callback** – An optional function or method with the signature `callback(link)` to be called when the link is closed.

`CURVE = 'Curve25519'`

The curve used for Elliptic Curve DH key exchanges

`ESTABLISHMENT_TIMEOUT_PER_HOP = 5`

Default timeout for link establishment in seconds per hop to destination.

`KEEPALIVE = 360`

Interval for sending keep-alive packets on established links in seconds.

`identify(identity)`

Identifies the initiator of the link to the remote peer. This can only happen once the link has been established, and is carried out over the encrypted link. The identity is only revealed to the remote peer, and initiator anonymity is thus preserved. This method can be used for authentication.

Parameters `identity` – An `RNS.Identity` instance to identify as.

`request(path, data=None, response_callback=None, failed_callback=None, progress_callback=None, timeout=None)`

Sends a request to the remote peer.

Parameters

- **path** – The request path.
- **response_callback** – An optional function or method with the signature `response_callback(request_receipt)` to be called when a response is received. See the [Request Example](#) for more info.
- **failed_callback** – An optional function or method with the signature `failed_callback(request_receipt)` to be called when a request fails. See the [Request Example](#) for more info.
- **progress_callback** – An optional function or method with the signature `progress_callback(request_receipt)` to be called when progress is made receiving

the response. Progress can be accessed as a float between 0.0 and 1.0 by the *request_receipt.progress* property.

- **timeout** – An optional timeout in seconds for the request. If *None* is supplied it will be calculated based on link RTT.

Returns A *RNS.RequestReceipt* instance if the request was sent, or *False* if it was not.

no_inbound_for()

Returns The time in seconds since last inbound packet on the link.

no_outbound_for()

Returns The time in seconds since last outbound packet on the link.

inactive_for()

Returns The time in seconds since activity on the link.

get_remote_identity()

Returns The identity of the remote peer, if it is known

teardown()

Closes the link and purges encryption keys. New keys will be used if a new link to the same destination is established.

set_packet_callback(callback)

Registers a function to be called when a packet has been received over this link.

Parameters callback – A function or method with the signature *callback(message, packet)* to be called.

set_resource_callback(callback)

Registers a function to be called when a resource has been advertised over this link. If the function returns *True* the resource will be accepted. If it returns *False* it will be ignored.

Parameters callback – A function or method with the signature *callback(resource)* to be called.

set_resource_started_callback(callback)

Registers a function to be called when a resource has begun transferring over this link.

Parameters callback – A function or method with the signature *callback(resource)* to be called.

set_resource_concluded_callback(callback)

Registers a function to be called when a resource has concluded transferring over this link.

Parameters callback – A function or method with the signature *callback(resource)* to be called.

set_remote_identified_callback(callback)

Registers a function to be called when an initiating peer has identified over this link.

Parameters callback – A function or method with the signature *callback(identity)* to be called.

set_resource_strategy(resource_strategy)

Sets the resource strategy for the link.

Parameters resource_strategy – One of *RNS.Link.ACCEPT_NONE*, *RNS.Link.ACCEPT_ALL* or *RNS.Link.ACCEPT_APP*. If *RNS.Link.ACCEPT_APP* is set, the *re-*

source_callback will be called to determine whether the resource should be accepted or not.

Raises *TypeError* if the resource strategy is unsupported.

7.1.7 Request Receipt

class `RNS.RequestReceipt`

An instance of this class is returned by the `request` method of `RNS.Link` instances. It should never be instantiated manually. It provides methods to check status, response time and response data when the request concludes.

`get_request_id()`

Returns The request ID as *bytes*.

`get_status()`

Returns The current status of the request, one of `RNS.RequestReceipt.FAILED`, `RNS.RequestReceipt.SENT`, `RNS.RequestReceipt.DELIVERED`, `RNS.RequestReceipt.READY`.

`get_progress()`

Returns The progress of a response being received as a *float* between 0.0 and 1.0.

`get_response()`

Returns The response as *bytes* if it is ready, otherwise *None*.

`get_response_time()`

Returns The response time of the request in seconds.

7.1.8 Resource

class `RNS.Resource`(*data, link, advertise=True, auto_compress=True, callback=None, progress_callback=None, timeout=None*)

The Resource class allows transferring arbitrary amounts of data over a link. It will automatically handle sequencing, compression, coordination and checksumming.

Parameters

- **data** – The data to be transferred. Can be *bytes* or an open *file handle*. See the *Filetransfer Example* for details.
- **link** – The *RNS.Link* instance on which to transfer the data.
- **advertise** – Optional. Whether to automatically advertise the resource. Can be *True* or *False*.
- **auto_compress** – Optional. Whether to auto-compress the resource. Can be *True* or *False*.
- **callback** – An optional *callable* with the signature *callback(resource)*. Will be called when the resource transfer concludes.

- **progress_callback** – An optional *callable* with the signature *callback(resource)*. Will be called whenever the resource transfer progress is updated.

advertise()

Advertise the resource. If the other end of the link accepts the resource advertisement it will begin transferring.

cancel()

Cancels transferring the resource.

get_progress()

Returns The current progress of the resource transfer as a *float* between 0.0 and 1.0.

7.1.9 Transport

class RNS.Transport

Through static methods of this class you can interact with the Transport system of Reticulum.

PATHFINDER_M = 128

Maximum amount of hops that Reticulum will transport a packet.

static register_announce_handler(handler)

Registers an announce handler.

Parameters handler – Must be an object with an *aspect_filter* attribute and a *received_announce(destination_hash, announced_identity, app_data)* callable. See the *Announce Example* for more info.

static deregister_announce_handler(handler)

Deregisters an announce handler.

Parameters handler – The announce handler to be deregistered.

static has_path(destination_hash)

Parameters destination_hash – A destination hash as *bytes*.

Returns *True* if a path to the destination is known, otherwise *False*.

static hops_to(destination_hash)

Parameters destination_hash – A destination hash as *bytes*.

Returns The number of hops to the specified destination, or `RNS.Transport.PATHFINDER_M` if the number of hops is unknown.

static next_hop(destination_hash)

Parameters destination_hash – A destination hash as *bytes*.

Returns The destination hash as *bytes* for the next hop to the specified destination, or *None* if the next hop is unknown.

static next_hop_interface(destination_hash)

Parameters destination_hash – A destination hash as *bytes*.

Returns The interface for the next hop to the specified destination, or *None* if the interface is unknown.

static request_path(*destination_hash*, *on_interface=None*)

Requests a path to the destination from the network. If another reachable peer on the network knows a path, it will announce it.

Parameters

- **destination_hash** – A destination hash as *bytes*.
- **on_interface** – If specified, the path request will only be sent on this interface. In normal use, Reticulum handles this automatically, and this parameter should not be used.

CODE EXAMPLES

A number of examples are included in the source distribution of Reticulum. You can use these examples to learn how to write your own programs.

8.1 Minimal

The *Minimal* example demonstrates the bare-minimum setup required to connect to a Reticulum network from your program. In about five lines of code, you will have the Reticulum Network Stack initialised, and ready to pass traffic in your program.

```
#####  
# This RNS example demonstrates a minimal setup, that #  
# will start up the Reticulum Network Stack, generate a #  
# new destination, and let the user send an announce. #  
#####  
  
import argparse  
import RNS  
  
# Let's define an app name. We'll use this for all  
# destinations we create. Since this basic example  
# is part of a range of example utilities, we'll put  
# them all within the app namespace "example_utilities"  
APP_NAME = "example_utilities"  
  
# This initialisation is executed when the program is started  
def program_setup(configpath):  
    # We must first initialise Reticulum  
    reticulum = RNS.Reticulum(configpath)  
  
    # Randomly create a new identity for our example  
    identity = RNS.Identity()  
  
    # Using the identity we just created, we create a destination.  
    # Destinations are endpoints in Reticulum, that can be addressed  
    # and communicated with. Destinations can also announce their  
    # existence, which will let the network know they are reachable  
    # and automatically create paths to them, from anywhere else  
    # in the network.  
    destination = RNS.Destination(  

```

(continues on next page)

(continued from previous page)

```

    identity,
    RNS.Destination.IN,
    RNS.Destination.SINGLE,
    APP_NAME,
    "minimalsample"
)

# We configure the destination to automatically prove all
# packets adressed to it. By doing this, RNS will automatically
# generate a proof for each incoming packet and transmit it
# back to the sender of that packet. This will let anyone that
# tries to communicate with the destination know whether their
# communication was received correctly.
destination.set_proof_strategy(RNS.Destination.PROVE_ALL)

# Everything's ready!
# Let's hand over control to the announce loop
announceLoop(destination)

def announceLoop(destination):
    # Let the user know that everything is ready
    RNS.log(
        "Minimal example "+
        RNS.prettyhexrep(destination.hash)+
        " running, hit enter to manually send an announce (Ctrl-C to quit)"
    )

    # We enter a loop that runs until the users exits.
    # If the user hits enter, we will announce our server
    # destination on the network, which will let clients
    # know how to create messages directed towards it.
    while True:
        entered = input()
        destination.announce()
        RNS.log("Sent announce from "+RNS.prettyhexrep(destination.hash))

#####
#### Program Startup #####
#####

# This part of the program gets run at startup,
# and parses input from the user, and then starts
# the desired program mode.
if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(
            description="Minimal example to start Reticulum and create a destination"
        )

        parser.add_argument(

```

(continues on next page)

(continued from previous page)

```

        "--config",
        action="store",
        default=None,
        help="path to alternative Reticulum config directory",
        type=str
    )

    args = parser.parse_args()

    if args.config:
        configarg = args.config
    else:
        configarg = None

    program_setup(configarg)

except KeyboardInterrupt:
    print("")
    exit()

```

This example can also be found at <https://github.com/markqvist/Reticulum/blob/master/Examples/Minimal.py>.

8.2 Announce

The *Announce* example builds upon the previous example by exploring how to announce a destination on the network, and how to let your program receive notifications about announces from relevant destinations.

```

#####
# This RNS example demonstrates setting up announce      #
# callbacks, which will let an application receive a    #
# notification when an announce relevant for it arrives #
#####

import argparse
import random
import RNS

# Let's define an app name. We'll use this for all
# destinations we create. Since this basic example
# is part of a range of example utilities, we'll put
# them all within the app namespace "example_utilities"
APP_NAME = "example_utilities"

# We initialise two lists of strings to use as app_data
fruits = ["Peach", "Quince", "Date", "Tangerine", "Pomelo", "Carambola", "Grape"]
noble_gases = ["Helium", "Neon", "Argon", "Krypton", "Xenon", "Radon", "Oganesson"]

# This initialisation is executed when the program is started
def program_setup(configpath):
    # We must first initialise Reticulum

```

(continues on next page)

```
reticulum = RNS.Reticulum(configpath)

# Randomly create a new identity for our example
identity = RNS.Identity()

# Using the identity we just created, we create two destinations
# in the "example_utilities.announcesample" application space.
#
# Destinations are endpoints in Reticulum, that can be addressed
# and communicated with. Destinations can also announce their
# existence, which will let the network know they are reachable
# and automatically create paths to them, from anywhere else
# in the network.
destination_1 = RNS.Destination(
    identity,
    RNS.Destination.IN,
    RNS.Destination.SINGLE,
    APP_NAME,
    "announcesample",
    "fruits"
)

destination_2 = RNS.Destination(
    identity,
    RNS.Destination.IN,
    RNS.Destination.SINGLE,
    APP_NAME,
    "announcesample",
    "noble_gases"
)

# We configure the destinations to automatically prove all
# packets addressed to it. By doing this, RNS will automatically
# generate a proof for each incoming packet and transmit it
# back to the sender of that packet. This will let anyone that
# tries to communicate with the destination know whether their
# communication was received correctly.
destination_1.set_proof_strategy(RNS.Destination.PROVE_ALL)
destination_2.set_proof_strategy(RNS.Destination.PROVE_ALL)

# We create an announce handler and configure it to only ask for
# announces from "example_utilities.announcesample.fruits".
# Try changing the filter and see what happens.
announce_handler = ExampleAnnounceHandler(
    aspect_filter="example_utilities.announcesample.fruits"
)

# We register the announce handler with Reticulum
RNS.Transport.register_announce_handler(announce_handler)

# Everything's ready!
# Let's hand over control to the announce loop
```

(continues on next page)

(continued from previous page)

```

announceLoop(destination_1, destination_2)

def announceLoop(destination_1, destination_2):
    # Let the user know that everything is ready
    RNS.log("Announce example running, hit enter to manually send an announce (Ctrl-C to_
↳quit)")

    # We enter a loop that runs until the users exits.
    # If the user hits enter, we will announce our server
    # destination on the network, which will let clients
    # know how to create messages directed towards it.
    while True:
        entered = input()

        # Randomly select a fruit
        fruit = fruits[random.randint(0, len(fruits)-1)]

        # Send the announce including the app data
        destination_1.announce(app_data=fruit.encode("utf-8"))
        RNS.log(
            "Sent announce from "+
            RNS.prettyhexrep(destination_1.hash)+
            " (" +destination_1.name+")"
        )

        # Randomly select a noble gas
        noble_gas = noble_gases[random.randint(0, len(noble_gases)-1)]

        # Send the announce including the app data
        destination_2.announce(app_data=noble_gas.encode("utf-8"))
        RNS.log(
            "Sent announce from "+
            RNS.prettyhexrep(destination_2.hash)+
            " (" +destination_2.name+")"
        )

# We will need to define an announce handler class that
# Reticulum can message when an announce arrives.
class ExampleAnnounceHandler:
    # The initialisation method takes the optional
    # aspect_filter argument. If aspect_filter is set to
    # None, all announces will be passed to the instance.
    # If only some announces are wanted, it can be set to
    # an aspect string.
    def __init__(self, aspect_filter=None):
        self.aspect_filter = aspect_filter

    # This method will be called by Reticulum's Transport
    # system when an announce arrives that matches the
    # configured aspect filter. Filters must be specific,
    # and cannot use wildcards.

```

(continues on next page)

```

def received_announce(self, destination_hash, announced_identity, app_data):
    RNS.log(
        "Received an announce from "+
        RNS.prettyhexrep(destination_hash)
    )

    RNS.log(
        "The announce contained the following app data: "+
        app_data.decode("utf-8")
    )

#####
#### Program Startup #####
#####

# This part of the program gets run at startup,
# and parses input from the user, and then starts
# the desired program mode.
if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(
            description="Reticulum example that demonstrates announces and announce_
↳handlers"
        )

        parser.add_argument(
            "--config",
            action="store",
            default=None,
            help="path to alternative Reticulum config directory",
            type=str
        )

        args = parser.parse_args()

        if args.config:
            configarg = args.config
        else:
            configarg = None

        program_setup(configarg)

    except KeyboardInterrupt:
        print("")
        exit()

```

This example can also be found at <https://github.com/markqvist/Reticulum/blob/master/Examples/Announce.py>.

8.3 Broadcast

The *Broadcast* example explores how to transmit plaintext broadcast messages over the network.

```
#####
# This RNS example demonstrates broadcasting unencrypted #
# information to any listening destinations.             #
#####

import sys
import argparse
import RNS

# Let's define an app name. We'll use this for all
# destinations we create. Since this basic example
# is part of a range of example utilities, we'll put
# them all within the app namespace "example_utilities"
APP_NAME = "example_utilities"

# This initialisation is executed when the program is started
def program_setup(configpath, channel=None):
    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # If the user did not select a "channel" we use
    # a default one called "public_information".
    # This "channel" is added to the destination name-
    # space, so the user can select different broadcast
    # channels.
    if channel == None:
        channel = "public_information"

    # We create a PLAIN destination. This is an unencrypted endpoint
    # that anyone can listen to and send information to.
    broadcast_destination = RNS.Destination(
        None,
        RNS.Destination.IN,
        RNS.Destination.PLAIN,
        APP_NAME,
        "broadcast",
        channel
    )

    # We specify a callback that will get called every time
    # the destination receives data.
    broadcast_destination.set_packet_callback(packet_callback)

    # Everything's ready!
    # Let's hand over control to the main loop
    broadcastLoop(broadcast_destination)

def packet_callback(data, packet):
    # Simply print out the received data
```

(continues on next page)

(continued from previous page)

```

print("")
print("Received data: "+data.decode("utf-8")+"\r\n> ", end="")
sys.stdout.flush()

def broadcastLoop(destination):
    # Let the user know that everything is ready
    RNS.log(
        "Broadcast example "+
        RNS.prettyhexrep(destination.hash)+
        " running, enter text and hit enter to broadcast (Ctrl-C to quit)"
    )

    # We enter a loop that runs until the users exits.
    # If the user hits enter, we will send the information
    # that the user entered into the prompt.
    while True:
        print("> ", end="")
        entered = input()

        if entered != "":
            data = entered.encode("utf-8")
            packet = RNS.Packet(destination, data)
            packet.send()

#####
### Program Startup #####
#####

# This part of the program gets run at startup,
# and parses input from the user, and then starts
# the program.
if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(
            description="Reticulum example demonstrating sending and receiving broadcasts
↪"
        )

        parser.add_argument(
            "--config",
            action="store",
            default=None,
            help="path to alternative Reticulum config directory",
            type=str
        )

        parser.add_argument(
            "--channel",
            action="store",
            default=None,

```

(continues on next page)

(continued from previous page)

```

        help="broadcast channel name",
        type=str
    )

    args = parser.parse_args()

    if args.config:
        configarg = args.config
    else:
        configarg = None

    if args.channel:
        channelarg = args.channel
    else:
        channelarg = None

    program_setup(configarg, channelarg)

except KeyboardInterrupt:
    print("")
    exit()

```

This example can also be found at <https://github.com/markqvist/Reticulum/blob/master/Examples/Broadcast.py>.

8.4 Echo

The *Echo* example demonstrates communication between two destinations using the Packet interface.

```

#####
# This RNS example demonstrates a simple client/server #
# echo utility. A client can send an echo request to the #
# server, and the server will respond by proving receipt #
# of the packet. #
#####

import argparse
import RNS

# Let's define an app name. We'll use this for all
# destinations we create. Since this echo example
# is part of a range of example utilities, we'll put
# them all within the app namespace "example_utilities"
APP_NAME = "example_utilities"

#####
#### Server Part #####
#####

# This initialisation is executed when the users chooses

```

(continues on next page)

(continued from previous page)

```

# to run as a server
def server(configpath):
    global reticulum

    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Randomly create a new identity for our echo server
    server_identity = RNS.Identity()

    # We create a destination that clients can query. We want
    # to be able to verify echo replies to our clients, so we
    # create a "single" destination that can receive encrypted
    # messages. This way the client can send a request and be
    # certain that no-one else than this destination was able
    # to read it.
    echo_destination = RNS.Destination(
        server_identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "echo",
        "request"
    )

    # We configure the destination to automatically prove all
    # packets adressed to it. By doing this, RNS will automatically
    # generate a proof for each incoming packet and transmit it
    # back to the sender of that packet.
    echo_destination.set_proof_strategy(RNS.Destination.PROVE_ALL)

    # Tell the destination which function in our program to
    # run when a packet is received. We do this so we can
    # print a log message when the server receives a request
    echo_destination.set_packet_callback(server_callback)

    # Everything's ready!
    # Let's Wait for client requests or user input
    announceLoop(echo_destination)

def announceLoop(destination):
    # Let the user know that everything is ready
    RNS.log(
        "Echo server "+
        RNS.prettyhexrep(destination.hash)+
        " running, hit enter to manually send an announce (Ctrl-C to quit)"
    )

    # We enter a loop that runs until the users exits.
    # If the user hits enter, we will announce our server
    # destination on the network, which will let clients

```

(continues on next page)

(continued from previous page)

```

# know how to create messages directed towards it.
while True:
    entered = input()
    destination.announce()
    RNS.log("Sent announce from "+RNS.prettyhexrep(destination.hash))

def server_callback(message, packet):
    global reticulum

    # Tell the user that we received an echo request, and
    # that we are going to send a reply to the requester.
    # Sending the proof is handled automatically, since we
    # set up the destination to prove all incoming packets.

    reception_stats = ""
    if reticulum.is_connected_to_shared_instance:
        reception_rssi = reticulum.get_packet_rssi(packet.packet_hash)
        reception_snr = reticulum.get_packet_snr(packet.packet_hash)

        if reception_rssi != None:
            reception_stats += " [RSSI "+str(reception_rssi)+" dBm]"

        if reception_snr != None:
            reception_stats += " [SNR "+str(reception_snr)+" dBm]"

    else:
        if packet.rssi != None:
            reception_stats += " [RSSI "+str(packet.rssi)+" dBm]"

        if packet.snr != None:
            reception_stats += " [SNR "+str(packet.snr)+" dB]"

    RNS.log("Received packet from echo client, proof sent"+reception_stats)

#####
#### Client Part #####
#####

# This initialisation is executed when the users chooses
# to run as a client
def client(destination_hexhash, configpath, timeout=None):
    global reticulum

    # We need a binary representation of the destination
    # hash that was entered on the command line
    try:
        if len(destination_hexhash) != 20:
            raise ValueError(
                "Destination length is invalid, must be 20 hexadecimal characters (10
↳bytes)"

```

(continues on next page)

```

    )

    destination_hash = bytes.fromhex(destination_hexhash)
except:
    RNS.log("Invalid destination entered. Check your input!\n")
    exit()

# We must first initialise Reticulum
reticulum = RNS.Reticulum(configpath)

# We override the loglevel to provide feedback when
# an announce is received
if RNS.loglevel < RNS.LOG_INFO:
    RNS.loglevel = RNS.LOG_INFO

# Tell the user that the client is ready!
RNS.log(
    "Echo client ready, hit enter to send echo request to "+
    destination_hexhash+
    " (Ctrl-C to quit)"
)

# We enter a loop that runs until the user exits.
# If the user hits enter, we will try to send an
# echo request to the destination specified on the
# command line.
while True:
    input()

    # Let's first check if RNS knows a path to the destination.
    # If it does, we'll load the server identity and create a packet
    if RNS.Transport.has_path(destination_hash):

        # To address the server, we need to know it's public
        # key, so we check if Reticulum knows this destination.
        # This is done by calling the "recall" method of the
        # Identity module. If the destination is known, it will
        # return an Identity instance that can be used in
        # outgoing destinations.
        server_identity = RNS.Identity.recall(destination_hash)

        # We got the correct identity instance from the
        # recall method, so let's create an outgoing
        # destination. We use the naming convention:
        # example_utilities.echo.request
        # This matches the naming we specified in the
        # server part of the code.
        request_destination = RNS.Destination(
            server_identity,
            RNS.Destination.OUT,
            RNS.Destination.SINGLE,
            APP_NAME,

```

(continues on next page)

(continued from previous page)

```

        "echo",
        "request"
    )

    # The destination is ready, so let's create a packet.
    # We set the destination to the request_destination
    # that was just created, and the only data we add
    # is a random hash.
    echo_request = RNS.Packet(request_destination, RNS.Identity.get_random_
↪hash())

    # Send the packet! If the packet is successfully
    # sent, it will return a PacketReceipt instance.
    packet_receipt = echo_request.send()

    # If the user specified a timeout, we set this
    # timeout on the packet receipt, and configure
    # a callback function, that will get called if
    # the packet times out.
    if timeout != None:
        packet_receipt.set_timeout(timeout)
        packet_receipt.set_timeout_callback(packet_timed_out)

    # We can then set a delivery callback on the receipt.
    # This will get automatically called when a proof for
    # this specific packet is received from the destination.
    packet_receipt.set_delivery_callback(packet_delivered)

    # Tell the user that the echo request was sent
    RNS.log("Sent echo request to "+RNS.prettyhexrep(request_destination.hash))
else:
    # If we do not know this destination, tell the
    # user to wait for an announce to arrive.
    RNS.log("Destination is not yet known. Requesting path...")
    RNS.Transport.request_path(destination_hash)

# This function is called when our reply destination
# receives a proof packet.
def packet_delivered(receipt):
    global reticulum

    if receipt.status == RNS.PacketReceipt.DELIVERED:
        rtt = receipt.get_rtt()
        if (rtt >= 1):
            rtt = round(rtt, 3)
            rttstring = str(rtt)+" seconds"
        else:
            rtt = round(rtt*1000, 3)
            rttstring = str(rtt)+" milliseconds"

        reception_stats = ""
        if reticulum.is_connected_to_shared_instance:

```

(continues on next page)

(continued from previous page)

```

reception_rssi = reticulum.get_packet_rssi(receipt.proof_packet.packet_hash)
reception_snr  = reticulum.get_packet_snr(receipt.proof_packet.packet_hash)

if reception_rssi != None:
    reception_stats += " [RSSI "+str(reception_rssi)+" dBm]"

if reception_snr != None:
    reception_stats += " [SNR "+str(reception_snr)+" dB]"

else:
    if receipt.proof_packet != None:
        if receipt.proof_packet.rssi != None:
            reception_stats += " [RSSI "+str(receipt.proof_packet.rssi)+" dBm]"

            if receipt.proof_packet.snr != None:
                reception_stats += " [SNR "+str(receipt.proof_packet.snr)+" dB]"

RNS.log(
    "Valid reply received from "+
    RNS.prettyhexrep(receipt.destination.hash)+
    ", round-trip time is "+rttstring+
    reception_stats
)

# This function is called if a packet times out.
def packet_timed_out(receipt):
    if receipt.status == RNS.PacketReceipt.FAILED:
        RNS.log("Packet "+RNS.prettyhexrep(receipt.hash)+" timed out")

#####
#### Program Startup #####
#####

# This part of the program gets run at startup,
# and parses input from the user, and then starts
# the desired program mode.
if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(description="Simple echo server and client_
↳utility")

        parser.add_argument(
            "-s",
            "--server",
            action="store_true",
            help="wait for incoming packets from clients"
        )

        parser.add_argument(
            "-t",
            "--timeout",

```

(continues on next page)

(continued from previous page)

```
        action="store",
        metavar="s",
        default=None,
        help="set a reply timeout in seconds",
        type=float
    )

    parser.add_argument("--config",
        action="store",
        default=None,
        help="path to alternative Reticulum config directory",
        type=str
    )

    parser.add_argument(
        "destination",
        nargs="?",
        default=None,
        help="hexadecimal hash of the server destination",
        type=str
    )

    args = parser.parse_args()

    if args.server:
        configarg=None
        if args.config:
            configarg = args.config
        server(configarg)
    else:
        if args.config:
            configarg = args.config
        else:
            configarg = None

        if args.timeout:
            timeoutarg = float(args.timeout)
        else:
            timeoutarg = None

        if (args.destination == None):
            print("")
            parser.print_help()
            print("")
        else:
            client(args.destination, configarg, timeout=timeoutarg)
    except KeyboardInterrupt:
        print("")
        exit()
```

This example can also be found at <https://github.com/markqvist/Reticulum/blob/master/Examples/Echo.py>.

8.5 Link

The *Link* example explores establishing an encrypted link to a remote destination, and passing traffic back and forth over the link.

```
#####
# This RNS example demonstrates how to set up a link to #
# a destination, and pass data back and forth over it. #
#####

import os
import sys
import time
import argparse
import RNS

# Let's define an app name. We'll use this for all
# destinations we create. Since this echo example
# is part of a range of example utilities, we'll put
# them all within the app namespace "example_utilities"
APP_NAME = "example_utilities"

#####
#### Server Part #####
#####

# A reference to the latest client link that connected
latest_client_link = None

# This initialisation is executed when the users chooses
# to run as a server
def server(configpath):
    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Randomly create a new identity for our link example
    server_identity = RNS.Identity()

    # We create a destination that clients can connect to. We
    # want clients to create links to this destination, so we
    # need to create a "single" destination type.
    server_destination = RNS.Destination(
        server_identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "linkexample"
    )

    # We configure a function that will get called every time
    # a new client creates a link to this destination.
    server_destination.set_link_established_callback(client_connected)
```

(continues on next page)

(continued from previous page)

```

# Everything's ready!
# Let's Wait for client requests or user input
server_loop(server_destination)

def server_loop(destination):
    # Let the user know that everything is ready
    RNS.log(
        "Link example "+
        RNS.prettyhexrep(destination.hash)+
        " running, waiting for a connection."
    )

    RNS.log("Hit enter to manually send an announce (Ctrl-C to quit)")

    # We enter a loop that runs until the users exits.
    # If the user hits enter, we will announce our server
    # destination on the network, which will let clients
    # know how to create messages directed towards it.
    while True:
        entered = input()
        destination.announce()
        RNS.log("Sent announce from "+RNS.prettyhexrep(destination.hash))

# When a client establishes a link to our server
# destination, this function will be called with
# a reference to the link.
def client_connected(link):
    global latest_client_link

    RNS.log("Client connected")
    link.set_link_closed_callback(client_disconnected)
    link.set_packet_callback(server_packet_received)
    latest_client_link = link

def client_disconnected(link):
    RNS.log("Client disconnected")

def server_packet_received(message, packet):
    global latest_client_link

    # When data is received over any active link,
    # it will all be directed to the last client
    # that connected.
    text = message.decode("utf-8")
    RNS.log("Received data on the link: "+text)

    reply_text = "I received \""+text+"\" over the link"
    reply_data = reply_text.encode("utf-8")
    RNS.Packet(latest_client_link, reply_data).send()

```

#####

(continues on next page)

(continued from previous page)

```

#### Client Part #####
#####

# A reference to the server link
server_link = None

# This initialisation is executed when the users chooses
# to run as a client
def client(destination_hexhash, configpath):
    # We need a binary representation of the destination
    # hash that was entered on the command line
    try:
        if len(destination_hexhash) != 20:
            raise ValueError("Destination length is invalid, must be 20 hexadecimal_
↳characters (10 bytes)")
        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        exit()

    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Check if we know a path to the destination
    if not RNS.Transport.has_path(destination_hash):
        RNS.log("Destination is not yet known. Requesting path and waiting for announce_
↳to arrive...")
        RNS.Transport.request_path(destination_hash)
        while not RNS.Transport.has_path(destination_hash):
            time.sleep(0.1)

    # Recall the server identity
    server_identity = RNS.Identity.recall(destination_hash)

    # Inform the user that we'll begin connecting
    RNS.log("Establishing link with server...")

    # When the server identity is known, we set
    # up a destination
    server_destination = RNS.Destination(
        server_identity,
        RNS.Destination.OUT,
        RNS.Destination.SINGLE,
        APP_NAME,
        "linkexample"
    )

    # And create a link
    link = RNS.Link(server_destination)

    # We set a callback that will get executed
    # every time a packet is received over the

```

(continues on next page)

(continued from previous page)

```

# link
link.set_packet_callback(client_packet_received)

# We'll also set up functions to inform the
# user when the link is established or closed
link.set_link_established_callback(link_established)
link.set_link_closed_callback(link_closed)

# Everything is set up, so let's enter a loop
# for the user to interact with the example
client_loop()

def client_loop():
    global server_link

    # Wait for the link to become active
    while not server_link:
        time.sleep(0.1)

    should_quit = False
    while not should_quit:
        try:
            print("> ", end=" ")
            text = input()

            # Check if we should quit the example
            if text == "quit" or text == "q" or text == "exit":
                should_quit = True
                server_link.teardown()

            # If not, send the entered text over the link
            if text != "":
                data = text.encode("utf-8")
                if len(data) <= RNS.Link.MDU:
                    RNS.Packet(server_link, data).send()
                else:
                    RNS.log(
                        "Cannot send this packet, the data size of "+
                        str(len(data))+ " bytes exceeds the link packet MDU of "+
                        str(RNS.Link.MDU)+ " bytes",
                        RNS.LOG_ERROR
                    )

        except Exception as e:
            RNS.log("Error while sending data over the link: "+str(e))
            should_quit = True
            server_link.teardown()

# This function is called when a link
# has been established with the server
def link_established(link):
    # We store a reference to the link

```

(continues on next page)

(continued from previous page)

```

# instance for later use
global server_link
server_link = link

# Inform the user that the server is
# connected
RNS.log("Link established with server, enter some text to send, or \"quit\" to quit")

# When a link is closed, we'll inform the
# user, and exit the program
def link_closed(link):
    if link.teardown_reason == RNS.Link.TIMEOUT:
        RNS.log("The link timed out, exiting now")
    elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
        RNS.log("The link was closed by the server, exiting now")
    else:
        RNS.log("Link closed, exiting now")

    RNS.Reticulum.exit_handler()
    time.sleep(1.5)
    os._exit(0)

# When a packet is received over the link, we
# simply print out the data.
def client_packet_received(message, packet):
    text = message.decode("utf-8")
    RNS.log("Received data on the link: "+text)
    print("> ", end=" ")
    sys.stdout.flush()

#####
#### Program Startup #####
#####

# This part of the program runs at startup,
# and parses input of from the user, and then
# starts up the desired program mode.
if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(description="Simple link example")

        parser.add_argument(
            "-s",
            "--server",
            action="store_true",
            help="wait for incoming link requests from clients"
        )

        parser.add_argument(
            "--config",
            action="store",

```

(continues on next page)

(continued from previous page)

```

        default=None,
        help="path to alternative Reticulum config directory",
        type=str
    )

    parser.add_argument(
        "destination",
        nargs="?",
        default=None,
        help="hexadecimal hash of the server destination",
        type=str
    )

    args = parser.parse_args()

    if args.config:
        configarg = args.config
    else:
        configarg = None

    if args.server:
        server(configarg)
    else:
        if (args.destination == None):
            print("")
            parser.print_help()
            print("")
        else:
            client(args.destination, configarg)

    except KeyboardInterrupt:
        print("")
        exit()

```

This example can also be found at <https://github.com/markqvist/Reticulum/blob/master/Examples/Link.py>.

8.6 Identification

The *Identify* example explores identifying an initiator of a link, once the link has been established.

```

#####
# This RNS example demonstrates how to set up a link to #
# a destination, and identify the initiator to it's peer #
#####

import os
import sys
import time
import argparse
import RNS

```

(continues on next page)

(continued from previous page)

```

# Let's define an app name. We'll use this for all
# destinations we create. Since this echo example
# is part of a range of example utilities, we'll put
# them all within the app namespace "example_utilities"
APP_NAME = "example_utilities"

#####
#### Server Part #####
#####

# A reference to the latest client link that connected
latest_client_link = None

# This initialisation is executed when the users chooses
# to run as a server
def server(configpath):
    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Randomly create a new identity for our link example
    server_identity = RNS.Identity()

    # We create a destination that clients can connect to. We
    # want clients to create links to this destination, so we
    # need to create a "single" destination type.
    server_destination = RNS.Destination(
        server_identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "identifyexample"
    )

    # We configure a function that will get called every time
    # a new client creates a link to this destination.
    server_destination.set_link_established_callback(client_connected)

    # Everything's ready!
    # Let's Wait for client requests or user input
    server_loop(server_destination)

def server_loop(destination):
    # Let the user know that everything is ready
    RNS.log(
        "Link identification example "+
        RNS.prettyhexrep(destination.hash)+
        " running, waiting for a connection."
    )

    RNS.log("Hit enter to manually send an announce (Ctrl-C to quit)")

```

(continues on next page)

(continued from previous page)

```

# We enter a loop that runs until the users exits.
# If the user hits enter, we will announce our server
# destination on the network, which will let clients
# know how to create messages directed towards it.
while True:
    entered = input()
    destination.announce()
    RNS.log("Sent announce from "+RNS.prettyhexrep(destination.hash))

# When a client establishes a link to our server
# destination, this function will be called with
# a reference to the link.
def client_connected(link):
    global latest_client_link

    RNS.log("Client connected")
    link.set_link_closed_callback(client_disconnected)
    link.set_packet_callback(server_packet_received)
    link.set_remote_identified_callback(remote_identified)
    latest_client_link = link

def client_disconnected(link):
    RNS.log("Client disconnected")

def remote_identified(identity):
    RNS.log("Remote identified as: "+str(identity))

def server_packet_received(message, packet):
    global latest_client_link

    # Get the originating identity for display
    remote_peer = "unidentified peer"
    if packet.link.get_remote_identity() != None:
        remote_peer = str(packet.link.get_remote_identity())

    # When data is received over any active link,
    # it will all be directed to the last client
    # that connected.
    text = message.decode("utf-8")

    RNS.log("Received data from "+remote_peer+": "+text)

    reply_text = "I received \""+text+"\" over the link from "+remote_peer
    reply_data = reply_text.encode("utf-8")
    RNS.Packet(latest_client_link, reply_data).send()

#####
#### Client Part #####
#####

# A reference to the server link

```

(continues on next page)

(continued from previous page)

```

server_link = None

# A reference to the client identity
client_identity = None

# This initialisation is executed when the users chooses
# to run as a client
def client(destination_hexhash, configpath):
    global client_identity
    # We need a binary representation of the destination
    # hash that was entered on the command line
    try:
        if len(destination_hexhash) != 20:
            raise ValueError("Destination length is invalid, must be 20 hexadecimal_
↳characters (10 bytes)")
        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        exit()

    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Create a new client identity
    client_identity = RNS.Identity()
    RNS.log(
        "Client created new identity "+
        str(client_identity)
    )

    # Check if we know a path to the destination
    if not RNS.Transport.has_path(destination_hash):
        RNS.log("Destination is not yet known. Requesting path and waiting for announce_
↳to arrive...")
        RNS.Transport.request_path(destination_hash)
        while not RNS.Transport.has_path(destination_hash):
            time.sleep(0.1)

    # Recall the server identity
    server_identity = RNS.Identity.recall(destination_hash)

    # Inform the user that we'll begin connecting
    RNS.log("Establishing link with server...")

    # When the server identity is known, we set
    # up a destination
    server_destination = RNS.Destination(
        server_identity,
        RNS.Destination.OUT,
        RNS.Destination.SINGLE,
        APP_NAME,
        "identifyexample"

```

(continues on next page)

(continued from previous page)

```

)

# And create a link
link = RNS.Link(server_destination)

# We set a callback that will get executed
# every time a packet is received over the
# link
link.set_packet_callback(client_packet_received)

# We'll also set up functions to inform the
# user when the link is established or closed
link.set_link_established_callback(link_established)
link.set_link_closed_callback(link_closed)

# Everything is set up, so let's enter a loop
# for the user to interact with the example
client_loop()

def client_loop():
    global server_link

    # Wait for the link to become active
    while not server_link:
        time.sleep(0.1)

    should_quit = False
    while not should_quit:
        try:
            print("> ", end=" ")
            text = input()

            # Check if we should quit the example
            if text == "quit" or text == "q" or text == "exit":
                should_quit = True
                server_link.teardown()

            # If not, send the entered text over the link
            if text != "":
                data = text.encode("utf-8")
                if len(data) <= RNS.Link.MDU:
                    RNS.Packet(server_link, data).send()
                else:
                    RNS.log(
                        "Cannot send this packet, the data size of "+
                        str(len(data))+ " bytes exceeds the link packet MDU of "+
                        str(RNS.Link.MDU)+ " bytes",
                        RNS.LOG_ERROR
                    )

        except Exception as e:
            RNS.log("Error while sending data over the link: "+str(e))

```

(continues on next page)

(continued from previous page)

```

        should_quit = True
        server_link.teardown()

# This function is called when a link
# has been established with the server
def link_established(link):
    # We store a reference to the link
    # instance for later use
    global server_link, client_identity
    server_link = link

    # Inform the user that the server is
    # connected
    RNS.log("Link established with server, identifying to remote peer...")

    link.identify(client_identity)

# When a link is closed, we'll inform the
# user, and exit the program
def link_closed(link):
    if link.teardown_reason == RNS.Link.TIMEOUT:
        RNS.log("The link timed out, exiting now")
    elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
        RNS.log("The link was closed by the server, exiting now")
    else:
        RNS.log("Link closed, exiting now")

    RNS.Reticulum.exit_handler()
    time.sleep(1.5)
    os._exit(0)

# When a packet is received over the link, we
# simply print out the data.
def client_packet_received(message, packet):
    text = message.decode("utf-8")
    RNS.log("Received data on the link: "+text)
    print("> ", end=" ")
    sys.stdout.flush()

#####
#### Program Startup #####
#####

# This part of the program runs at startup,
# and parses input of from the user, and then
# starts up the desired program mode.
if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(description="Simple link example")

        parser.add_argument(

```

(continues on next page)

(continued from previous page)

```
        "-s",
        "--server",
        action="store_true",
        help="wait for incoming link requests from clients"
    )

    parser.add_argument(
        "--config",
        action="store",
        default=None,
        help="path to alternative Reticulum config directory",
        type=str
    )

    parser.add_argument(
        "destination",
        nargs="?",
        default=None,
        help="hexadecimal hash of the server destination",
        type=str
    )

    args = parser.parse_args()

    if args.config:
        configarg = args.config
    else:
        configarg = None

    if args.server:
        server(configarg)
    else:
        if (args.destination == None):
            print("")
            parser.print_help()
            print("")
        else:
            client(args.destination, configarg)

    except KeyboardInterrupt:
        print("")
        exit()
```

This example can also be found at <https://github.com/markqvist/Reticulum/blob/master/Examples/Identify.py>.

8.7 Requests & Responses

The *Request* example explores sending requests and receiving responses.

```
#####
# This RNS example demonstrates how to set perform      #
# requests and receive responses over a link.          #
#####

import os
import sys
import time
import random
import argparse
import RNS

# Let's define an app name. We'll use this for all
# destinations we create. Since this echo example
# is part of a range of example utilities, we'll put
# them all within the app namespace "example_utilities"
APP_NAME = "example_utilities"

#####
#### Server Part #####
#####

# A reference to the latest client link that connected
latest_client_link = None

def random_text_generator(path, data, request_id, remote_identity, requested_at):
    RNS.log("Generating response to request "+RNS.prettyhexrep(request_id))
    texts = ["They looked up", "On each full moon", "Becky was upset", "I'll stay away_
↳from it", "The pet shop stocks everything"]
    return texts[random.randint(0, len(texts)-1)]

# This initialisation is executed when the users chooses
# to run as a server
def server(configpath):
    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Randomly create a new identity for our link example
    server_identity = RNS.Identity()

    # We create a destination that clients can connect to. We
    # want clients to create links to this destination, so we
    # need to create a "single" destination type.
    server_destination = RNS.Destination(
        server_identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "requestexample"
```

(continues on next page)

(continued from previous page)

```

)

# We configure a function that will get called every time
# a new client creates a link to this destination.
server_destination.set_link_established_callback(client_connected)

# We register a request handler for handling incoming
# requests over any established links.
server_destination.register_request_handler(
    "/random/text",
    response_generator = random_text_generator,
    allow = RNS.Destination.ALLOW_ALL
)

# Everything's ready!
# Let's Wait for client requests or user input
server_loop(server_destination)

def server_loop(destination):
    # Let the user know that everything is ready
    RNS.log(
        "Request example "+
        RNS.prettyhexrep(destination.hash)+
        " running, waiting for a connection."
    )

    RNS.log("Hit enter to manually send an announce (Ctrl-C to quit)")

    # We enter a loop that runs until the users exits.
    # If the user hits enter, we will announce our server
    # destination on the network, which will let clients
    # know how to create messages directed towards it.
    while True:
        entered = input()
        destination.announce()
        RNS.log("Sent announce from "+RNS.prettyhexrep(destination.hash))

# When a client establishes a link to our server
# destination, this function will be called with
# a reference to the link.
def client_connected(link):
    global latest_client_link

    RNS.log("Client connected")
    link.set_link_closed_callback(client_disconnected)
    latest_client_link = link

def client_disconnected(link):
    RNS.log("Client disconnected")

```

#####

(continues on next page)

(continued from previous page)

```

##### Client Part #####
#####

# A reference to the server link
server_link = None

# This initialisation is executed when the users chooses
# to run as a client
def client(destination_hexhash, configpath):
    # We need a binary representation of the destination
    # hash that was entered on the command line
    try:
        if len(destination_hexhash) != 20:
            raise ValueError("Destination length is invalid, must be 20 hexadecimal_
↳characters (10 bytes)")
        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        exit()

    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Check if we know a path to the destination
    if not RNS.Transport.has_path(destination_hash):
        RNS.log("Destination is not yet known. Requesting path and waiting for announce_
↳to arrive...")
        RNS.Transport.request_path(destination_hash)
        while not RNS.Transport.has_path(destination_hash):
            time.sleep(0.1)

    # Recall the server identity
    server_identity = RNS.Identity.recall(destination_hash)

    # Inform the user that we'll begin connecting
    RNS.log("Establishing link with server...")

    # When the server identity is known, we set
    # up a destination
    server_destination = RNS.Destination(
        server_identity,
        RNS.Destination.OUT,
        RNS.Destination.SINGLE,
        APP_NAME,
        "requestexample"
    )

    # And create a link
    link = RNS.Link(server_destination)

    # We'll set up functions to inform the
    # user when the link is established or closed

```

(continues on next page)

(continued from previous page)

```

link.set_link_established_callback(link_established)
link.set_link_closed_callback(link_closed)

# Everything is set up, so let's enter a loop
# for the user to interact with the example
client_loop()

def client_loop():
    global server_link

    # Wait for the link to become active
    while not server_link:
        time.sleep(0.1)

    should_quit = False
    while not should_quit:
        try:
            print("> ", end=" ")
            text = input()

            # Check if we should quit the example
            if text == "quit" or text == "q" or text == "exit":
                should_quit = True
                server_link.teardown()

            else:
                server_link.request(
                    "/random/text",
                    data = None,
                    response_callback = got_response,
                    failed_callback = request_failed
                )

        except Exception as e:
            RNS.log("Error while sending request over the link: "+str(e))
            should_quit = True
            server_link.teardown()

def got_response(request_receipt):
    request_id = request_receipt.request_id
    response = request_receipt.response

    RNS.log("Got response for request "+RNS.prettyhexrep(request_id)+": "+str(response))

def request_received(request_receipt):
    RNS.log("The request "+RNS.prettyhexrep(request_receipt.request_id)+" was received,
    ↪by the remote peer.")

def request_failed(request_receipt):
    RNS.log("The request "+RNS.prettyhexrep(request_receipt.request_id)+" failed.")

```

(continues on next page)

(continued from previous page)

```

# This function is called when a link
# has been established with the server
def link_established(link):
    # We store a reference to the link
    # instance for later use
    global server_link
    server_link = link

    # Inform the user that the server is
    # connected
    RNS.log("Link established with server, hit enter to perform a request, or type in \
↪"quit\" to quit")

# When a link is closed, we'll inform the
# user, and exit the program
def link_closed(link):
    if link.teardown_reason == RNS.Link.TIMEOUT:
        RNS.log("The link timed out, exiting now")
    elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
        RNS.log("The link was closed by the server, exiting now")
    else:
        RNS.log("Link closed, exiting now")

    RNS.Reticulum.exit_handler()
    time.sleep(1.5)
    os._exit(0)

#####
#### Program Startup #####
#####

# This part of the program runs at startup,
# and parses input of from the user, and then
# starts up the desired program mode.
if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(description="Simple request/response example")

        parser.add_argument(
            "-s",
            "--server",
            action="store_true",
            help="wait for incoming requests from clients"
        )

        parser.add_argument(
            "--config",
            action="store",
            default=None,
            help="path to alternative Reticulum config directory",

```

(continues on next page)

(continued from previous page)

```

        type=str
    )

    parser.add_argument(
        "destination",
        nargs="?",
        default=None,
        help="hexadecimal hash of the server destination",
        type=str
    )

    args = parser.parse_args()

    if args.config:
        configarg = args.config
    else:
        configarg = None

    if args.server:
        server(configarg)
    else:
        if (args.destination == None):
            print("")
            parser.print_help()
            print("")
        else:
            client(args.destination, configarg)

    except KeyboardInterrupt:
        print("")
        exit()

```

This example can also be found at <https://github.com/markqvist/Reticulum/blob/master/Examples/Request.py>.

8.8 Filetransfer

The *Filetransfer* example implements a basic file-server program that allow clients to connect and download files. The program uses the Resource interface to efficiently pass files of any size over a Reticulum *Link*.

```

#####
# This RNS example demonstrates a simple filetransfer #
# server and client program. The server will serve a #
# directory of files, and the clients can list and #
# download files from the server. #
# #
# Please note that using RNS Resources for large file #
# transfers is not recommended, since compression, #
# encryption and hashmap sequencing can take a long time #
# on systems with slow CPUs, which will probably result #
# in the client timing out before the resource sender #

```

(continues on next page)

(continued from previous page)

```

# can complete preparing the resource.          #
#                                                #
# If you need to transfer large files, use the Bundle #
# class instead, which will automatically slice the data #
# into chunks suitable for packing as a Resource.     #
#####

import os
import sys
import time
import threading
import argparse
import RNS
import RNS.vendor.umsgpack as umsgpack

# Let's define an app name. We'll use this for all
# destinations we create. Since this echo example
# is part of a range of example utilities, we'll put
# them all within the app namespace "example_utilities"
APP_NAME = "example_utilities"

# We'll also define a default timeout, in seconds
APP_TIMEOUT = 45.0

#####
#### Server Part #####
#####

serve_path = None

# This initialisation is executed when the users chooses
# to run as a server
def server(configpath, path):
    # We must first initialise Reticulum
    reticulum = RNS.Reticulum(configpath)

    # Randomly create a new identity for our file server
    server_identity = RNS.Identity()

    global serve_path
    serve_path = path

    # We create a destination that clients can connect to. We
    # want clients to create links to this destination, so we
    # need to create a "single" destination type.
    server_destination = RNS.Destination(
        server_identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "filetransfer",
        "server"

```

(continues on next page)

(continued from previous page)

```

)

# We configure a function that will get called every time
# a new client creates a link to this destination.
server_destination.set_link_established_callback(client_connected)

# Everything's ready!
# Let's Wait for client requests or user input
announceLoop(server_destination)

def announceLoop(destination):
    # Let the user know that everything is ready
    RNS.log("File server "+RNS.prettyhexrep(destination.hash)+" running")
    RNS.log("Hit enter to manually send an announce (Ctrl-C to quit)")

    # We enter a loop that runs until the users exits.
    # If the user hits enter, we will announce our server
    # destination on the network, which will let clients
    # know how to create messages directed towards it.
    while True:
        entered = input()
        destination.announce()
        RNS.log("Sent announce from "+RNS.prettyhexrep(destination.hash))

# Here's a convenience function for listing all files
# in our served directory
def list_files():
    # We add all entries from the directory that are
    # actual files, and does not start with "."
    global serve_path
    return [file for file in os.listdir(serve_path) if os.path.isfile(os.path.join(serve_
↵path, file)) and file[:1] != "."]

# When a client establishes a link to our server
# destination, this function will be called with
# a reference to the link. We then send the client
# a list of files hosted on the server.
def client_connected(link):
    # Check if the served directory still exists
    if os.path.isdir(serve_path):
        RNS.log("Client connected, sending file list...")

        link.set_link_closed_callback(client_disconnected)

    # We pack a list of files for sending in a packet
    data = umsgpack.packb(list_files())

    # Check the size of the packed data
    if len(data) <= RNS.Link.MDU:
        # If it fits in one packet, we will just
        # send it as a single packet over the link.
        list_packet = RNS.Packet(link, data)

```

(continues on next page)

(continued from previous page)

```

        list_receipt = list_packet.send()
        list_receipt.set_timeout(APP_TIMEOUT)
        list_receipt.set_delivery_callback(list_delivered)
        list_receipt.set_timeout_callback(list_timeout)
    else:
        RNS.log("Too many files in served directory!", RNS.LOG_ERROR)
        RNS.log("You should implement a function to split the filelist over multiple_
↳packets.", RNS.LOG_ERROR)
        RNS.log("Hint: The client already supports it :)", RNS.LOG_ERROR)

        # After this, we're just going to keep the link
        # open until the client requests a file. We'll
        # configure a function that get's called when
        # the client sends a packet with a file request.
        link.set_packet_callback(client_request)
    else:
        RNS.log("Client connected, but served path no longer exists!", RNS.LOG_ERROR)
        link.teardown()

def client_disconnected(link):
    RNS.log("Client disconnected")

def client_request(message, packet):
    global serve_path

    try:
        filename = message.decode("utf-8")
    except Exception as e:
        filename = None

    if filename in list_files():
        try:
            # If we have the requested file, we'll
            # read it and pack it as a resource
            RNS.log("Client requested \""+filename+"\"")
            file = open(os.path.join(serve_path, filename), "rb")

            file_resource = RNS.Resource(
                file,
                packet.link,
                callback=resource_sending_concluded
            )

            file_resource.filename = filename
        except Exception as e:
            # If somethign went wrong, we close
            # the link
            RNS.log("Error while reading file \""+filename+"\"", RNS.LOG_ERROR)
            packet.link.teardown()
            raise e
    else:
        # If we don't have it, we close the link

```

(continues on next page)

(continued from previous page)

```

        RNS.log("Client requested an unknown file")
        packet.link.teardown()

# This function is called on the server when a
# resource transfer concludes.
def resource_sending_concluded(resource):
    if hasattr(resource, "filename"):
        name = resource.filename
    else:
        name = "resource"

    if resource.status == RNS.Resource.COMPLETE:
        RNS.log("Done sending \""+name+"\" to client")
    elif resource.status == RNS.Resource.FAILED:
        RNS.log("Sending \""+name+"\" to client failed")

def list_delivered(receipt):
    RNS.log("The file list was received by the client")

def list_timeout(receipt):
    RNS.log("Sending list to client timed out, closing this link")
    link = receipt.destination
    link.teardown()

#####
#### Client Part #####
#####

# We store a global list of files available on the server
server_files = []

# A reference to the server link
server_link = None

# And a reference to the current download
current_download = None
current_filename = None

# Variables to store download statistics
download_started = 0
download_finished = 0
download_time = 0
transfer_size = 0
file_size = 0

# This initialisation is executed when the users chooses
# to run as a client
def client(destination_hexhash, configpath):
    # We need a binary representation of the destination
    # hash that was entered on the command line
    try:

```

(continues on next page)

(continued from previous page)

```
        if len(destination_hexhash) != 20:
            raise ValueError("Destination length is invalid, must be 20 hexadecimal_
↳characters (10 bytes)")
            destination_hash = bytes.fromhex(destination_hexhash)
        except:
            RNS.log("Invalid destination entered. Check your input!\n")
            exit()

        # We must first initialise Reticulum
        reticulum = RNS.Reticulum(configpath)

        # Check if we know a path to the destination
        if not RNS.Transport.has_path(destination_hash):
            RNS.log("Destination is not yet known. Requesting path and waiting for announce_
↳to arrive...")
            RNS.Transport.request_path(destination_hash)
            while not RNS.Transport.has_path(destination_hash):
                time.sleep(0.1)

        # Recall the server identity
        server_identity = RNS.Identity.recall(destination_hash)

        # Inform the user that we'll begin connecting
        RNS.log("Establishing link with server...")

        # When the server identity is known, we set
        # up a destination
        server_destination = RNS.Destination(
            server_identity,
            RNS.Destination.OUT,
            RNS.Destination.SINGLE,
            APP_NAME,
            "filetransfer",
            "server"
        )

        # We also want to automatically prove incoming packets
        server_destination.set_proof_strategy(RNS.Destination.PROVE_ALL)

        # And create a link
        link = RNS.Link(server_destination)

        # We expect any normal data packets on the link
        # to contain a list of served files, so we set
        # a callback accordingly
        link.set_packet_callback(filelist_received)

        # We'll also set up functions to inform the
        # user when the link is established or closed
        link.set_link_established_callback(link_established)
        link.set_link_closed_callback(link_closed)
```

(continues on next page)

(continued from previous page)

```

# And set the link to automatically begin
# downloading advertised resources
link.set_resource_strategy(RNS.Link.ACCEPT_ALL)
link.set_resource_started_callback(download_began)
link.set_resource_concluded_callback(download_concluded)

menu()

# Requests the specified file from the server
def download(filename):
    global server_link, menu_mode, current_filename, transfer_size, download_started
    current_filename = filename
    download_started = 0
    transfer_size = 0

    # We just create a packet containing the
    # requested filename, and send it down the
    # link. We also specify we don't need a
    # packet receipt.
    request_packet = RNS.Packet(server_link, filename.encode("utf-8"), create_
↪receipt=False)
    request_packet.send()

    print("")
    print(("Requested \""+filename+"\" from server, waiting for download to begin.."))
    menu_mode = "download_started"

# This function runs a simple menu for the user
# to select which files to download, or quit
menu_mode = None
def menu():
    global server_files, server_link
    # Wait until we have a filelist
    while len(server_files) == 0:
        time.sleep(0.1)
    RNS.log("Ready!")
    time.sleep(0.5)

    global menu_mode
    menu_mode = "main"
    should_quit = False
    while (not should_quit):
        print_menu()

        while not menu_mode == "main":
            # Wait
            time.sleep(0.25)

        user_input = input()
        if user_input == "q" or user_input == "quit" or user_input == "exit":
            should_quit = True

```

(continues on next page)

(continued from previous page)

```

        print("")
    else:
        if user_input in server_files:
            download(user_input)
        else:
            try:
                if 0 <= int(user_input) < len(server_files):
                    download(server_files[int(user_input)])
            except:
                pass

    if should_quit:
        server_link.teardown()

# Prints out menus or screens for the
# various states of the client program.
# It's simple and quite uninteresting.
# I won't go into detail here. Just
# strings basically.
def print_menu():
    global menu_mode, download_time, download_started, download_finished, transfer_size, ↵
    ↵file_size

    if menu_mode == "main":
        clear_screen()
        print_filelist()
        print("")
        print("Select a file to download by entering name or number, or q to quit")
        print(("> "), end=' ')
    elif menu_mode == "download_started":
        download_began = time.time()
        while menu_mode == "download_started":
            time.sleep(0.1)
            if time.time() > download_began+APP_TIMEOUT:
                print("The download timed out")
                time.sleep(1)
                server_link.teardown()

    if menu_mode == "downloading":
        print("Download started")
        print("")
        while menu_mode == "downloading":
            global current_download
            percent = round(current_download.get_progress() * 100.0, 1)
            print(("rProgress: "+str(percent)+" %  "), end=' ')
            sys.stdout.flush()
            time.sleep(0.1)

    if menu_mode == "save_error":
        print(("rProgress: 100.0 %"), end=' ')
        sys.stdout.flush()
        print("")

```

(continues on next page)

(continued from previous page)

```

print("Could not write downloaded file to disk")
current_download.status = RNS.Resource.FAILED
menu_mode = "download_concluded"

if menu_mode == "download_concluded":
    if current_download.status == RNS.Resource.COMPLETE:
        print("\rProgress: 100.0 %", end=' ')
        sys.stdout.flush()

        # Print statistics
        hours, rem = divmod(download_time, 3600)
        minutes, seconds = divmod(rem, 60)
        timestring = "{:0>2}:{:0>2}:{:05.2f}".format(int(hours),int(minutes),seconds)
        print("")
        print("")
        print("--- Statistics -----")
        print("\tTime taken      : "+timestring)
        print("\tFile size       : "+size_str(file_size))
        print("\tData transferred : "+size_str(transfer_size))
        print("\tEffective rate   : "+size_str(file_size/download_time, suffix='b')+
↪"/s")
        print("\tTransfer rate    : "+size_str(transfer_size/download_time, suffix='b'
↪')+"/s")
        print("")
        print("The download completed! Press enter to return to the menu.")
        print("")
        input()

    else:
        print("")
        print("The download failed! Press enter to return to the menu.")
        input()

    current_download = None
    menu_mode = "main"
    print_menu()

# This function prints out a list of files
# on the connected server.
def print_filelist():
    global server_files
    print("Files on server:")
    for index,file in enumerate(server_files):
        print("\t("+str(index)+")\t"+file)

def filelist_received(filelist_data, packet):
    global server_files, menu_mode
    try:
        # Unpack the list and extend our
        # local list of available files
        filelist = umsgpack.unpackb(filelist_data)
        for file in filelist:

```

(continues on next page)

(continued from previous page)

```

        if not file in server_files:
            server_files.append(file)

        # If the menu is already visible,
        # we'll update it with what was
        # just received
        if menu_mode == "main":
            print_menu()
    except:
        RNS.log("Invalid file list data received, closing link")
        packet.link.teardown()

# This function is called when a link
# has been established with the server
def link_established(link):
    # We store a reference to the link
    # instance for later use
    global server_link
    server_link = link

    # Inform the user that the server is
    # connected
    RNS.log("Link established with server")
    RNS.log("Waiting for filelist...")

    # And set up a small job to check for
    # a potential timeout in receiving the
    # file list
    thread = threading.Thread(target=filelist_timeout_job)
    thread.setDaemon(True)
    thread.start()

# This job just sleeps for the specified
# time, and then checks if the file list
# was received. If not, the program will
# exit.
def filelist_timeout_job():
    time.sleep(APP_TIMEOUT)

    global server_files
    if len(server_files) == 0:
        RNS.log("Timed out waiting for filelist, exiting")
        os._exit(0)

# When a link is closed, we'll inform the
# user, and exit the program
def link_closed(link):
    if link.teardown_reason == RNS.Link.TIMEOUT:
        RNS.log("The link timed out, exiting now")
    elif link.teardown_reason == RNS.Link.DESTINATION_CLOSED:
        RNS.log("The link was closed by the server, exiting now")

```

(continues on next page)

(continued from previous page)

```

else:
    RNS.log("Link closed, exiting now")

RNS.Reticulum.exit_handler()
time.sleep(1.5)
os._exit(0)

# When RNS detects that the download has
# started, we'll update our menu state
# so the user can be shown a progress of
# the download.
def download_began(resource):
    global menu_mode, current_download, download_started, transfer_size, file_size
    current_download = resource

    if download_started == 0:
        download_started = time.time()

    transfer_size += resource.size
    file_size = resource.total_size

    menu_mode = "downloading"

# When the download concludes, successfully
# or not, we'll update our menu state and
# inform the user about how it all went.
def download_concluded(resource):
    global menu_mode, current_filename, download_started, download_finished, download_
    ↪time
    download_finished = time.time()
    download_time = download_finished - download_started

    saved_filename = current_filename

    if resource.status == RNS.Resource.COMPLETE:
        counter = 0
        while os.path.isfile(saved_filename):
            counter += 1
            saved_filename = current_filename+"."+str(counter)

        try:
            file = open(saved_filename, "wb")
            file.write(resource.data.read())
            file.close()
            menu_mode = "download_concluded"
        except:
            menu_mode = "save_error"
    else:
        menu_mode = "download_concluded"

# A convenience function for printing a human-
# readable file size

```

(continues on next page)

(continued from previous page)

```

def size_str(num, suffix='B'):
    units = ['', 'Ki', 'Mi', 'Gi', 'Ti', 'Pi', 'Ei', 'Zi']
    last_unit = 'Yi'

    if suffix == 'b':
        num *= 8
        units = ['', 'K', 'M', 'G', 'T', 'P', 'E', 'Z']
        last_unit = 'Y'

    for unit in units:
        if abs(num) < 1024.0:
            return "%3.2f %s%s" % (num, unit, suffix)
        num /= 1024.0
    return "%.2f %s%s" % (num, last_unit, suffix)

# A convenience function for clearing the screen
def clear_screen():
    os.system('cls' if os.name=='nt' else 'clear')

#####
### Program Startup #####
#####

# This part of the program runs at startup,
# and parses input of from the user, and then
# starts up the desired program mode.
if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(
            description="Simple file transfer server and client utility"
        )

        parser.add_argument(
            "-s",
            "--serve",
            action="store",
            metavar="dir",
            help="serve a directory of files to clients"
        )

        parser.add_argument(
            "--config",
            action="store",
            default=None,
            help="path to alternative Reticulum config directory",
            type=str
        )

        parser.add_argument(
            "destination",
            nargs="?",
            default=None,

```

(continues on next page)

(continued from previous page)

```
        help="hexadecimal hash of the server destination",
        type=str
    )

    args = parser.parse_args()

    if args.config:
        configarg = args.config
    else:
        configarg = None

    if args.serve:
        if os.path.isdir(args.serve):
            server(configarg, args.serve)
        else:
            RNS.log("The specified directory does not exist")
    else:
        if (args.destination == None):
            print("")
            parser.print_help()
            print("")
        else:
            client(args.destination, configarg)

    except KeyboardInterrupt:
        print("")
        exit()
```

This example can also be found at <https://github.com/markqvist/Reticulum/blob/master/Examples/Filetransfer.py>.

A

advertise() (*RNS.Resource* method), 59
 announce() (*RNS.Destination* method), 53
 ANNOUNCE_CAP (*RNS.Reticulum* attribute), 49
 app_and_aspects_from_name() (*RNS.Destination* static method), 52

C

cancel() (*RNS.Resource* method), 59
 clear_default_app_data() (*RNS.Destination* method), 54
 create_keys() (*RNS.Destination* method), 54
 CURVE (*RNS.Identity* attribute), 50
 CURVE (*RNS.Link* attribute), 56

D

decrypt() (*RNS.Destination* method), 54
 decrypt() (*RNS.Identity* method), 52
 deregister_announce_handler() (*RNS.Transport* static method), 59
 deregister_request_handler() (*RNS.Destination* method), 54
 Destination (*class in RNS*), 52

E

encrypt() (*RNS.Destination* method), 54
 encrypt() (*RNS.Identity* method), 51
 ENCRYPTED_MDU (*RNS.Packet* attribute), 55
 ESTABLISHMENT_TIMEOUT_PER_HOP (*RNS.Link* attribute), 56

F

from_bytes() (*RNS.Identity* static method), 51
 from_file() (*RNS.Identity* static method), 51
 full_hash() (*RNS.Identity* static method), 50
 full_name() (*RNS.Destination* static method), 52

G

get_private_key() (*RNS.Destination* method), 54
 get_private_key() (*RNS.Identity* method), 51
 get_progress() (*RNS.RequestReceipt* method), 58

get_progress() (*RNS.Resource* method), 59
 get_public_key() (*RNS.Identity* method), 51
 get_random_hash() (*RNS.Identity* static method), 51
 get_remote_identity() (*RNS.Link* method), 57
 get_request_id() (*RNS.RequestReceipt* method), 58
 get_response() (*RNS.RequestReceipt* method), 58
 get_response_time() (*RNS.RequestReceipt* method), 58
 get_rtt() (*RNS.PacketReceipt* method), 55
 get_status() (*RNS.PacketReceipt* method), 55
 get_status() (*RNS.RequestReceipt* method), 58

H

has_path() (*RNS.Transport* static method), 59
 hash() (*RNS.Destination* static method), 53
 hash_from_name_and_identity() (*RNS.Destination* static method), 53
 hops_to() (*RNS.Transport* static method), 59

I

identify() (*RNS.Link* method), 56
 Identity (*class in RNS*), 50
 inactive_for() (*RNS.Link* method), 57

K

KEEPALIVE (*RNS.Link* attribute), 56
 KEYSIZE (*RNS.Identity* attribute), 50

L

Link (*class in RNS*), 56
 load_private_key() (*RNS.Destination* method), 54
 load_private_key() (*RNS.Identity* method), 51
 load_public_key() (*RNS.Identity* method), 51

M

MTU (*RNS.Reticulum* attribute), 49

N

next_hop() (*RNS.Transport* static method), 59
 next_hop_interface() (*RNS.Transport* static method), 59

no_inbound_for() (*RNS.Link* method), 57
no_outbound_for() (*RNS.Link* method), 57

P

Packet (*class in RNS*), 55
PacketReceipt (*class in RNS*), 55
PATHFINDER_M (*RNS.Transport* attribute), 59
PLAIN_MDU (*RNS.Packet* attribute), 55

R

recall() (*RNS.Identity* static method), 50
recall_app_data() (*RNS.Identity* static method), 50
register_announce_handler() (*RNS.Transport* static method), 59
register_request_handler() (*RNS.Destination* method), 53
request() (*RNS.Link* method), 56
request_path() (*RNS.Transport* static method), 60
RequestReceipt (*class in RNS*), 58
resend() (*RNS.Packet* method), 55
Resource (*class in RNS*), 58
Reticulum (*class in RNS*), 49

S

send() (*RNS.Packet* method), 55
set_default_app_data() (*RNS.Destination* method), 54
set_delivery_callback() (*RNS.PacketReceipt* method), 56
set_link_established_callback() (*RNS.Destination* method), 53
set_packet_callback() (*RNS.Destination* method), 53
set_packet_callback() (*RNS.Link* method), 57
set_proof_requested_callback() (*RNS.Destination* method), 53
set_proof_strategy() (*RNS.Destination* method), 53
set_remote_identified_callback() (*RNS.Link* method), 57
set_resource_callback() (*RNS.Link* method), 57
set_resource_concluded_callback() (*RNS.Link* method), 57
set_resource_started_callback() (*RNS.Link* method), 57
set_resource_strategy() (*RNS.Link* method), 57
set_timeout() (*RNS.PacketReceipt* method), 55
set_timeout_callback() (*RNS.PacketReceipt* method), 56
should_use_implicit_proof() (*RNS.Reticulum* static method), 50
sign() (*RNS.Destination* method), 54
sign() (*RNS.Identity* method), 52

T

teardown() (*RNS.Link* method), 57
to_file() (*RNS.Identity* method), 51
Transport (*class in RNS*), 59
transport_enabled() (*RNS.Reticulum* static method), 50
truncated_hash() (*RNS.Identity* static method), 50
TRUNCATED_HASHLENGTH (*RNS.Identity* attribute), 50

V

validate() (*RNS.Identity* method), 52