CHAPTER

THREE

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, containing the networking core, and a set of utility and daemon programs.

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.

When you have Reticulum installed, any program or application that uses Reticulum will automatically load and initialise Reticulum when it starts, if it is not already running.

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 already running 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 Configuration & Data

Reticulum stores all information that it needs to function in a single file-system directory. When Reticulum is started, it will look for a valid configuration directory in the following places:

- `/etc/reticulum`
- `~/.config/reticulum`
- `~/.reticulum`

If no existing configuration directory is found, the directory `~/.reticulum` is created, and the default configuration will be automatically created here. You can move it to one of the other locations if you wish.

It is also possible to use completely arbitrary configuration directories by specifying the relevant command-line parameters when running Reticulum-based programs. You can also run multiple separate Reticulum instances on the same physical system, either in isolation from each other, or connected together.

In most cases, a single physical system will only need to run one Reticulum instance. This can either be launched at boot, as a system service, or simply be brought up when a program needs it. In either case, any number of programs running on the same system will automatically share the same Reticulum instance, if the configuration allows for it, which it does by default.

The entire configuration of Reticulum is found in the `~/.reticulum/config` file. When Reticulum is first started on a new system, a basic, but fully functional configuration file is created. The default configuration looks like this:

```
# This is the default Reticulum config file.
# You should probably edit it to include any additional,
# interfaces and settings you might need.
```

(continues on next page)

(continued from previous page)

```
# Only the most basic options are included in this default
# configuration. To see a more verbose, and much longer,
# configuration example, you can run the command:
# rnsd --exampleconfig


[reticulum]

# If you enable Transport, your system will route traffic
# for other peers, pass announces and serve path requests.
# This should only be done for systems that are suited to
# act as transport nodes, ie. if they are stationary and
# always-on. This directive is optional and can be removed
# for brevity.

enable_transport = False


# By default, the first program to launch the Reticulum
# Network Stack will create a shared instance, that other
# programs can communicate with. Only the shared instance
# opens all the configured interfaces directly, and other
# local programs communicate with the shared instance over
# a local socket. This is completely transparent to the
# user, and should generally be turned on. This directive
# is optional and can be removed for brevity.

share_instance = Yes


# If you want to run multiple *different* shared instances
# on the same system, you will need to specify different
# shared instance ports for each. The defaults are given
# below, and again, these options can be left out if you
# don't need them.

shared_instance_port = 37428
instance_control_port = 37429


# On systems where running instances may not have access
# to the same shared Reticulum configuration directory,
# it is still possible to allow full interactivity for
# running instances, by manually specifying a shared RPC
# key. In almost all cases, this option is not needed, but
# it can be useful on operating systems such as Android.
# The key must be specified as bytes in hexadecimal.

# rpc_key = e5c032d3ec4e64a6aca9927ba8ab73336780f6d71790
```

(continues on next page)

(continued from previous page)

```
# You can configure Reticulum to panic and forcibly close
# if an unrecoverable interface error occurs, such as the
# hardware device for an interface disappearing. This is
# an optional directive, and can be left out for brevity.
# This behaviour is disabled by default.

panic_on_interface_error = No


# When Transport is enabled, it is possible to allow the
# Transport Instance to respond to probe requests from
# the rnprobe utility. This can be a useful tool to test
# connectivity. When this option is enabled, the probe
# destination will be generated from the Identity of the
# Transport Instance, and printed to the log at startup.
# Optional, and disabled by default.

respond_to_probes = No


[logging]
# Valid log levels are 0 through 7:
#   0: Log only critical information
#   1: Log errors and lower log levels
#   2: Log warnings and lower log levels
#   3: Log notices and lower log levels
#   4: Log info and lower (this is the default)
#   5: Verbose logging
#   6: Debug logging
#   7: Extreme logging

loglevel = 4


# The interfaces section defines the physical and virtual
# interfaces Reticulum will use to communicate on. This
# section will contain examples for a variety of interface
# types. You can modify these or use them as a basis for
# your own config, or simply remove the unused ones.

[interfaces]

  # This interface enables communication with other
  # link-local Reticulum nodes over UDP. It does not
  # need any functional IP infrastructure like routers
  # or DHCP servers, but will require that at least link-
  # local IPv6 is enabled in your operating system, which
  # should be enabled by default in almost any OS. See
  # the Reticulum Manual for more configuration options.

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

(continues on next page)

(continued from previous page)

```
    interface_enabled = True
```

If Reticulum infrastructure already exists locally, you probably don't need to change anything, and you may already be connected to a wider network. If not, you will probably need to add relevant *interfaces* to the configuration, in order to communicate with other systems.

You can generate a much more verbose configuration example by running the command:

```
rnsd --exampleconfig
```

The output includes examples for most interface types supported by Reticulum, along with additional options and configuration parameters.

It is a good idea to read the comments and explanations in the above default config. It will teach you the basic concepts you need to understand to configure your network. Once you have done that, take a look at the *Interfaces* chapter of this manual.

3.2 Included Utility Programs

Reticulum includes a range of useful utilities, both for managing your Reticulum networks, and for carrying out common tasks over Reticulum networks, such as transferring files to remote systems, and executing commands and programs remotely.

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.

3.2.1 The rnsd Utility

It is very easy to run Reticulum as a service. 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.

Usage Examples

Run `rnsd`:

```
$ rnsd

[2023-08-18 17:59:56] [Notice] Started rnsd version 0.5.8
```

Run `rnsd` in service mode, ensuring all logging output is sent directly to file:

```
$ rnsd -s
```

Generate a verbose and detailed configuration example, with explanations of all the various configuration options, and interface configuration examples:

```
$ rnsd --exampleconfig
```

All Command-Line Options

```
usage: rnsd.py [-h] [--config CONFIG] [-v] [-q] [-s] [--exampleconfig] [--version]

Reticulum Network Stack Daemon

options:
  -h, --help       show this help message and exit
  --config CONFIG  path to alternative Reticulum config directory
  -v, --verbose
  -q, --quiet
  -s, --service    rnsd is running as a service and should log to file
  --exampleconfig  print verbose configuration example to stdout and exit
  --version        show program's version number and exit
```

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

3.2.2 The rnstatus Utility

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

Usage Examples

Run `rnstatus`:

```
$ rnstatus

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 Dublin/dublin.connect.reticulum.network: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>
```

(continues on next page)

(continued from previous page)

```
   Traffic : 8.49 KB↑
             9.23 KB↓

Reticulum Transport Instance <5245a8efe1788c6a1cd36144a270e13b> running
```

Filter output to only show some interfaces:

```
$ rnstatus rnode

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 <5245a8efe1788c6a1cd36144a270e13b> running
```

All Command-Line Options

```
usage: rnstatus [-h] [--config CONFIG] [--version] [-a] [-A]
                [-l] [-s SORT] [-r] [-j] [-R hash] [-i path]
                [-w seconds] [-v] [filter]

Reticulum Network Stack Status

positional arguments:
  filter                only display interfaces with names including filter

options:
  -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
  -A, --announce-stats  show announce stats
  -l, --link-stats      show link stats
  -s SORT, --sort SORT  sort interfaces by [rate, traffic, rx, tx, announces, arx, atx,␣
↪held]
  -r, --reverse         reverse sorting
  -j, --json            output in JSON format
  -R hash               transport identity hash of remote instance to get status from
  -i path               path to identity used for remote management
  -w seconds            timeout before giving up on remote queries
  -v, --verbose
```

3.2.3 The rnid Utility

With the `rnid` utility, you can generate, manage and view Reticulum Identities. The program can also calculate Destination hashes, and perform encryption and decryption of files.

Using `rnid`, it is possible to asymmetrically encrypt files and information for any Reticulum destination hash, and also to create and verify cryptographic signatures.

Usage Examples

Generate a new Identity:

```
$ rnid -g ./new_identity
```

Display Identity key information:

```
$ rnid -i ./new_identity -p

Loaded Identity <984b74a3f768bef236af4371e6f248cd> from new_id
Public Key  : 0f4259fef4521ab75a3409e353fe9073eb10783b4912a6a9937c57bf44a62c1e
Private Key : Hidden
```

Encrypt a file for an LXMF user:

```
$ rnid -i 8dd57a7382268096460893335a6b03695 -e my_file.txt

Recalled Identity <bc7291552be7a58f361522990465165c> for destination
↪<8dd57a7382268096460893335a6b03695>
Encrypting my_file.txt
File my_file.txt encrypted for <bc7291552be7a58f361522990465165c> to my_file.txt.rfe
```

If the Identity for the destination is not already known, you can fetch it from the network by using the `-R` command-line option:

```
$ rnid -R -i 30602def3b3506a28ed33db6f60cc6c9 -e my_file.txt

Requesting unknown Identity for <30602def3b3506a28ed33db6f60cc6c9>...
Received Identity <2b489d06eaf7c543808c76a5332a447d> for destination
↪<30602def3b3506a28ed33db6f60cc6c9> from the network
Encrypting my_file.txt
File my_file.txt encrypted for <2b489d06eaf7c543808c76a5332a447d> to my_file.txt.rfe
```

Decrypt a file using the Reticulum Identity it was encrypted for:

```
$ rnid -i ./my_identity -d my_file.txt.rfe

Loaded Identity <2225fdeecaf6e2db4556c3c2d7637294> from ./my_identity
Decrypting ./my_file.txt.rfe...
File ./my_file.txt.rfe decrypted with <2225fdeecaf6e2db4556c3c2d7637294> to ./my_file.txt
```

All Command-Line Options

```
usage: rnid.py [-h] [--config path] [-i identity] [-g path] [-v] [-q] [-a aspects]
               [-H aspects] [-e path] [-d path] [-s path] [-V path] [-r path] [-w path]
               [-f] [-R] [-t seconds] [-p] [-P] [--version]
```

(continues on next page)

(continued from previous page)

```
Reticulum Identity & Encryption Utility

options:
  -h, --help            show this help message and exit
  --config path         path to alternative Reticulum config directory
  -i identity, --identity identity
                        hexadecimal Reticulum Destination hash or path to Identity file
  -g path, --generate path
                        generate a new Identity
  -v, --verbose         increase verbosity
  -q, --quiet           decrease verbosity
  -a aspects, --announce aspects
                        announce a destination based on this Identity
  -H aspects, --hash aspects
                        show destination hashes for other aspects for this Identity
  -e path, --encrypt path
                        encrypt file
  -d path, --decrypt path
                        decrypt file
  -s path, --sign path  sign file
  -V path, --validate path
                        validate signature
  -r path, --read path  input file path
  -w path, --write path
                        output file path
  -f, --force           write output even if it overwrites existing files
  -R, --request         request unknown Identities from the network
  -t seconds            identity request timeout before giving up
  -p, --print-identity  print identity info and exit
  -P, --print-private   allow displaying private keys
  --version             show program's version number and exit
```

3.2.4 The rnpath Utility

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

Usage Examples

Resolve path to a destination:

```
$ rnpath c89b4da064bf66d280f0e4d8abfd9806

Path found, destination <c89b4da064bf66d280f0e4d8abfd9806> is 4 hops away via
↪<f53a1c4278e0726bb73fcc623d6ce763> on TCPInterface[Testnet/dublin.connect.reticulum.
↪network:4965]
```

All Command-Line Options

```
usage: rnpath [-h] [--config CONFIG] [--version] [-t] [-m hops]
              [-r] [-d] [-D] [-x] [-w seconds] [-R hash] [-i path]
              [-W seconds] [-j] [-v] [destination]
```

(continues on next page)

(continued from previous page)

```
Reticulum Path Discovery Utility

positional arguments:
  destination           hexadecimal hash of the destination

options:
  -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
  -m hops, --max hops   maximum hops to filter path table by
  -r, --rates           show announce rate info
  -d, --drop            remove the path to a destination
  -D, --drop-announces  drop all queued announces
  -x, --drop-via        drop all paths via specified transport instance
  -w seconds            timeout before giving up
  -R hash               transport identity hash of remote instance to manage
  -i path               path to identity used for remote management
  -W seconds            timeout before giving up on remote queries
  -j, --json            output in JSON format
  -v, --verbose
```

3.2.5 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, so most destinations will not be probable.

You can enable a probe-reply destination on Reticulum Transport Instances by setting the `respond_to_probes` configuration directive. Reticulum will then print the probe destination to the log on Transport Instance startup.

Usage Examples

Probe a destination:

```
$ rnprobe rnstransport.probe 2d03725b327348980d570f739a3a5708

Sent 16 byte probe to <2d03725b327348980d570f739a3a5708>
Valid reply received from <2d03725b327348980d570f739a3a5708>
Round-trip time is 38.469 milliseconds over 2 hops
```

Send a larger probe:

```
$ rnprobe rnstransport.probe 2d03725b327348980d570f739a3a5708 -s 256

Sent 16 byte probe to <2d03725b327348980d570f739a3a5708>
Valid reply received from <2d03725b327348980d570f739a3a5708>
Round-trip time is 38.781 milliseconds over 2 hops
```

If the interface that receives the probe replies supports reporting radio parameters such as **RSSI** and **SNR**, the `rnprobe` utility will print these as part of the result as well.

```
$ rnprobe rnstransport.probe e7536ee90bd4a440e130490b87a25124

Sent 16 byte probe to <e7536ee90bd4a440e130490b87a25124>
Valid reply received from <e7536ee90bd4a440e130490b87a25124>
Round-trip time is 1.809 seconds over 1 hop [RSSI -73 dBm] [SNR 12.0 dB]
```

All Command-Line Options

```
usage: rnprobe [-h] [--config CONFIG] [-s SIZE] [-n PROBES]
               [-t seconds] [-w seconds] [--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

options:
  -h, --help            show this help message and exit
  --config CONFIG       path to alternative Reticulum config directory
  -s SIZE, --size SIZE  size of probe packet payload in bytes
  -n PROBES, --probes PROBES
                        number of probes to send
  -t seconds, --timeout seconds
                        timeout before giving up
  -w seconds, --wait seconds
                        time between each probe
  --version             show program's version number and exit
  -v, --verbose
```

3.2.6 The rncp Utility

The `rncp` utility is a simple file transfer tool. Using it, you can transfer files through Reticulum.

Usage Examples

Run rncp on the receiving system, specifying which identities are allowed to send files:

```
$ rncp --listen -a 1726dbad538775b5bf9b0ea25a4079c8 -a c50cc4e4f7838b6c31f60ab9032cbc62
```

You can also specify allowed identity hashes (one per line) in the file ~/.rncp/allowed_identities and simply running the program in listener mode:

```
$ rncp --listen
```

From another system, copy a file to the receiving system:

```
$ rncp ~/path/to/file.tgz 73cbd378bb0286ed11a707c13447bb1e
```

Or fetch a file from the remote system:

```
$ rncp --fetch ~/path/to/file.tgz 73cbd378bb0286ed11a707c13447bb1e
```

All Command-Line Options

```
usage: rncp [-h] [--config path] [-v] [-q] [-S] [-l] [-F] [-f]
            [-j path] [-b seconds] [-a allowed_hash] [-n] [-p]
            [-w seconds] [--version] [file] [destination]

Reticulum File Transfer Utility

positional arguments:
  file                  file to be transferred
  destination           hexadecimal hash of the receiver

options:
  -h, --help            show this help message and exit
  --config path         path to alternative Reticulum config directory
  -v, --verbose         increase verbosity
  -q, --quiet           decrease verbosity
  -S, --silent          disable transfer progress output
  -l, --listen          listen for incoming transfer requests
  -F, --allow-fetch     allow authenticated clients to fetch files
  -f, --fetch           fetch file from remote listener instead of sending
  -j path, --jail path  restrict fetch requests to specified path
  -b seconds            announce interval, 0 to only announce at startup
  -a allowed_hash       allow this identity
  -n, --no-auth         accept requests from anyone
  -p, --print-identity  print identity and destination info and exit
  -w seconds            sender timeout before giving up
  --version             show program's version number and exit
```

3.2.7 The rnx Utility

The `rnx` utility is a basic remote command execution program. It allows you to execute commands on remote systems over Reticulum, and to view returned command output. For a fully interactive remote shell solution, be sure to also take a look at the rnsh program.

Usage Examples

Run rnx on the listening system, specifying which identities are allowed to execute commands:

```
$ rnx --listen -a 941bed5e228775e5a8079fc38b1ccf3f -a 1b03013c25f1c2ca068a4f080b844a10
```

From another system, run a command on the remote:

```
$ rnx 7a55144adf826958a9529a3bcf08b149 "cat /proc/cpuinfo"
```

Or enter the interactive mode pseudo-shell:

```
$ rnx 7a55144adf826958a9529a3bcf08b149 -x
```

The default identity file is stored in `~/.reticulum/identities/rnx`, but you can use another one, which will be created if it does not already exist

```
$ rnx 7a55144adf826958a9529a3bcf08b149 -i /path/to/identity -x
```

All Command-Line Options

```
usage: rnx [-h] [--config path] [-v] [-q] [-p] [-l] [-i identity] [-x] [-b] [-n] [-N]
           [-d] [-m] [-a allowed_hash] [-w seconds] [-W seconds] [--stdin STDIN]
           [--stdout STDOUT] [--stderr STDERR] [--version] [destination] [command]

Reticulum Remote Execution Utility

positional arguments:
  destination           hexadecimal hash of the listener
  command               command to be execute

optional arguments:
  -h, --help            show this help message and exit
  --config path         path to alternative Reticulum config directory
  -v, --verbose         increase verbosity
  -q, --quiet           decrease verbosity
  -p, --print-identity  print identity and destination info and exit
  -l, --listen          listen for incoming commands
  -i identity           path to identity to use
  -x, --interactive     enter interactive mode
  -b, --no-announce     don't announce at program start
  -a allowed_hash       accept from this identity
  -n, --noauth          accept files from anyone
  -N, --noid            don't identify to listener
  -d, --detailed        show detailed result output
  -m                    mirror exit code of remote command
  -w seconds            connect and request timeout before giving up
  -W seconds            max result download time
  --stdin STDIN         pass input to stdin
  --stdout STDOUT       max size in bytes of returned stdout
  --stderr STDERR       max size in bytes of returned stderr
  --version             show program's version number and exit
```

3.2.8 The rnodeconf Utility

The `rnodeconf` utility allows you to inspect and configure existing *RNodes*, and to create and provision new *RNodes* from any supported hardware devices.

All Command-Line Options

```
usage: rnodeconf [-h] [-i] [-a] [-u] [-U] [--fw-version version]
                 [--fw-url url] [--nocheck] [-e] [-E] [-C]
                 [--baud-flash baud_flash] [-N] [-T] [-b] [-B] [-p] [-D i]
                 [--display-addr byte] [--freq Hz] [--bw Hz] [--txp dBm]
                 [--sf factor] [--cr rate] [--eeprom-backup] [--eeprom-dump]
                 [--eeprom-wipe] [-P] [--trust-key hexbytes] [--version] [-f]
                 [-r] [-k] [-S] [-H FIRMWARE_HASH] [--platform platform]
                 [--product product] [--model model] [--hwrev revision]
                 [port]

RNode Configuration and firmware utility. This program allows you to change
various settings and startup modes of RNode. It can also install, flash and
update the firmware on supported devices.
```

(continues on next page)

(continued from previous page)

```
positional arguments:
  port                  serial port where RNode is attached

options:
  -h, --help            show this help message and exit
  -i, --info            Show device info
  -a, --autoinstall     Automatic installation on various supported devices
  -u, --update          Update firmware to the latest version
  -U, --force-update    Update to specified firmware even if version matches
                        or is older than installed version
  --fw-version version  Use a specific firmware version for update or
                        autoinstall
  --fw-url url          Use an alternate firmware download URL
  --nocheck             Don't check for firmware updates online
  -e, --extract         Extract firmware from connected RNode for later use
  -E, --use-extracted   Use the extracted firmware for autoinstallation or
                        update
  -C, --clear-cache     Clear locally cached firmware files
  --baud-flash baud_flash
                        Set specific baud rate when flashing device. Default
                        is 921600
  -N, --normal          Switch device to normal mode
  -T, --tnc             Switch device to TNC mode
  -b, --bluetooth-on    Turn device bluetooth on
  -B, --bluetooth-off   Turn device bluetooth off
  -p, --bluetooth-pair  Put device into bluetooth pairing mode
  -D i, --display i     Set display intensity (0-255)
  --display-addr byte   Set display address as hex byte (00 - FF)
  --freq Hz             Frequency in Hz for TNC mode
  --bw Hz               Bandwidth in Hz for TNC mode
  --txp dBm             TX power in dBm for TNC mode
  --sf factor           Spreading factor for TNC mode (7 - 12)
  --cr rate             Coding rate for TNC mode (5 - 8)
  --eeprom-backup       Backup EEPROM to file
  --eeprom-dump         Dump EEPROM to console
  --eeprom-wipe         Unlock and wipe EEPROM
  -P, --public          Display public part of signing key
  --trust-key hexbytes  Public key to trust for device verification
  --version             Print program version and exit
  -f, --flash           Flash firmware and bootstrap EEPROM
  -r, --rom             Bootstrap EEPROM without flashing firmware
  -k, --key             Generate a new signing key and exit
  -S, --sign            Display public part of signing key
  -H FIRMWARE_HASH, --firmware-hash FIRMWARE_HASH
                        Display installed firmware hash
  --platform platform   Platform specification for device bootstrap
  --product product     Product specification for device bootstrap
  --model model         Model code for device bootstrap
  --hwrev revision      Hardware revision for device bootstrap
```

For more information on how to create your own RNodes, please read the *Creating RNodes* section of this manual.

3.3 Remote Management

It is possible to allow remote management of Reticulum systems using the various built-in utilities, such as `rnstatus` and `rnpath`. To do so, you will need to set the `enable_remote_management` directive in the `[reticulum]` section of the configuration file. You will also need to specify one or more Reticulum Identity hashes for authenticating the queries from client programs. For this purpose, you can use existing identity files, or generate new ones with the rnid utility.

The following is a truncated example of enabling remote management in the Reticulum configuration file:

```
[reticulum]
...
enable_remote_management = yes
remote_management_allowed = 9fb6d773498fb3feda407ed8ef2c3229,␣
↪2d882c5586e548d79b5af27bca1776dc
...
```

For a complete example configuration, you can run `rnsd --exampleconfig`.

3.4 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.

3.4.1 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.

3.4.2 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.

Systemwide Service

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
```

Userspace Service

Alternatively you can use a user systemd service instead of a system wide one. This way the whole setup can be done as a regular user. Create a user systemd service files `~/.config/systemd/user/rnsd.service` with the following content:

```
[Unit]
Description=Reticulum Network Stack Daemon
After=default.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
ExecStart=RNS_BIN_DIR/rnsd --service

[Install]
WantedBy=default.target
```

Replace `RNS_BIN_DIR` with the path to your Reticulum binary directory (eg. /home/USERNAMEHERE/rns/bin).

Start user service:

```
systemctl --user daemon-reload
systemctl --user start rnsd.service
```

If you want to automatically start `rnsd` without having to log in as the USERNAMEHERE, do:

```
sudo loginctl enable-linger USERNAMEHERE
systemctl --user enable rnsd.service
```

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*.

4.2 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 *5 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.

4.3 Introduction & Basic Functionality

Reticulum is a networking stack suited for high-latency, low-bandwidth links. Reticulum is at its core a *message oriented* system. It is suited for both local point-to-point or point-to-multipoint scenarios where all 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 its 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 16 byte hash. This hash is derived from truncating a full SHA-256 hash of identifying characteristics of the destination. To users, the destination addresses will be displayed as 16 hexadecimal bytes, like this example: <13425ec15b621c1d928589718000d814>.

The truncation size of 16 bytes (128 bits) for destinations has been chosen as a reasonable trade-off 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 128 bits should be sufficient, even far into the future.

By default Reticulum encrypts all data using elliptic curve cryptography and AES. Any packet sent to a destination is encrypted with a per-packet derived key. Reticulum can also set up an encrypted channel to a destination, called a *Link*. Both data sent over Links and single packets offer *Initiator Anonymity*. Links additionally offer *Forward Secrecy* by default, employing an Elliptic Curve Diffie Hellman key exchange on Curve25519 to derive per-link ephemeral keys. Asymmetric, link-less packet communication can also provide forward secrecy, with automatic key ratcheting, by enabling ratchets on a per-destination basis. 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 local broadcast purposes.

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.

4.3.1 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 128 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 a 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      : 4faf1b2e0a077e6a9d92fa051f256038
```

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 appending 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 destinations 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.

4.3.2 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 its 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.

4.3.3 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.

4.3.4 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*.

4.4 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 it's destination.

4.4.1 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`.

4.4.2 The Announce Mechanism in Detail

When an *announce* for a destination is transmitted by 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.

4.4.3 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 destination's public key (or ratchet key, if available), 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.
- 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 its 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 destination's 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 can 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 297 bytes (more info in the *Binary Packet Format* section). The amount of bandwidth used on keeping a link open is practically negligible, at 0.45 bits per second. Even

on a slow 1200 bits per second packet radio channel, 100 concurrent links will still leave 96% 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 it's *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 *original 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. Packets can now be exchanged bi-directionally from either end of the link simply by adressing the packets to the *link id* of the link.
- 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 it's 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.

4.4.4 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.

4.5 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 5 bits per second or greater, 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.

4.6 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.

4.6.1 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.

4.6.2 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.

4.6.3 Wire Format

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

A Reticulum packet is composed of the following fields:

[HEADER 2 bytes] [ADDRESSES 16/32 bytes] [CONTEXT 1 byte] [DATA 0-465 bytes]

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

* Interface Access Code field if the IFAC flag was set.
```

(continues on next page)

(continued from previous page)

```
  * 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 16 bytes long.
  * The Header Type flag in the HEADER field determines
    whether the ADDRESSES field contains 1 or 2 addresses.
  * Addresses are SHA-256 hashes truncated to 16 bytes.

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

* The DATA field is between 0 and 465 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 16 byte address field
type 2           1  Two byte header, two 16 byte address fields


Context Flag
-----------------
unset            0  The context flag is used for various types
set              1  of signalling, depending on packet context


Propagation Types
-----------------
broadcast        0
transport        1


Destination Types
-----------------
single          00
group           01
plain           10
link            11


Packet Types
-----------------
data            00
announce        01
```

(continues on next page)

(continued from previous page)

```
link request    10
proof           11


+- Packet Example -+

   HEADER FIELD           DESTINATION FIELDS            CONTEXT FIELD  DATA FIELD
 _______|_______   _________________|_________________   _________|______   __|_
|               | |                                   | |               | |    |
01010000 00000100 [HASH1, 16 bytes] [HASH2, 16 bytes] [CONTEXT, 1 byte] [DATA]
|| | | |    |
|| | | |    +-- Hops             = 4
|| | | +------- Packet Type      = DATA
|| | +--------- 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, 16 bytes] [CONTEXT, 1 byte] [DATA]
|| | | |    |
|| | | |    +-- Hops             = 7
|| | | +------- 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, 16 bytes] [CONTEXT, 1 byte] [DATA]
|| | | |    |
|| | | |    +-- Hops             = 7
|| | | +------- 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
---------------------------------------
```

(continues on next page)

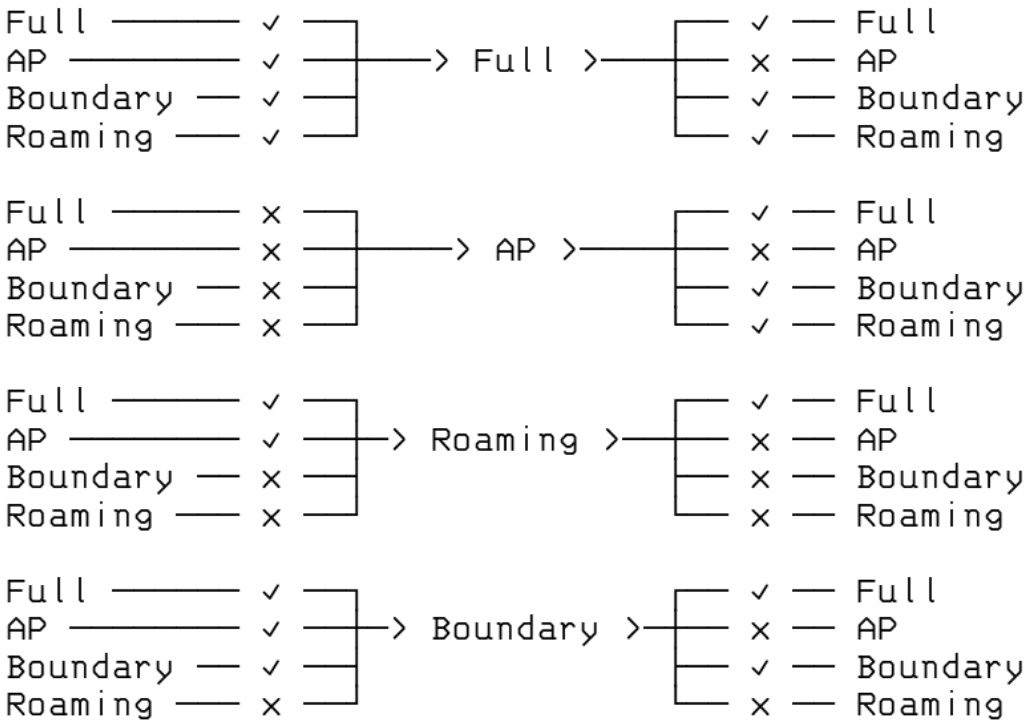
(continued from previous page)

```
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    :    51  bytes
- Announce        :    167 bytes
- Link Request    :    83  bytes
- Link Proof      :    115 bytes
- Link RTT packet :    99  bytes
- Link keepalive  :    20  bytes
```

4.6.4 Announce Propagation Rules

The following table illustrates the rules for automatically propagating announces from one interface type to another, for all possible combinations. For the purpose of announce propagation, the *Full* and *Gateway* modes are identical.



See the *Interface Modes* section for a conceptual overview of the different interface modes, and how they are configured.

CHAPTER

SIX

CONFIGURING 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.

6.1 Custom Interfaces

In addition to the built-in interface types, Reticulum is **fully extensible** with custom, user- or community-supplied interfaces, and creating custom interface modules is straightforward. Please see the *custom interface* example for basic interface code to build upon.

6.2 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 bare-minimum setup
# of an Auto Interface. It will allow communica-
# tion with all other reachable devices on all
# usable physical ethernet-based devices that
# are available on the system.

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

# This example demonstrates an more specifically
# configured Auto Interface, that only uses spe-
# cific physical interfaces, and has a number of
# other configuration options set.
```

(continues on next page)

(continued from previous page)

```
[[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 choose the multicast address type:
  # temporary (default, Temporary Multicast Address)
  # or permanent (Permanent Multicast Address)

  multicast_address_type = permanent

  # 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
  data_port = 49555
```

6.3 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 = 5urvjicpzi7q3ybztsef4i5ow2aq4soktfj7zedz53s47r54jnqq.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.

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.

6.4 TCP Server Interface

The TCP Server interface is suitable for allowing other peers to connect over the Internet or private IPv4 and IPv6 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
```

(continues on next page)

(continued from previous page)

```
# 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
```

If you are using the interface on a device which has both IPv4 and IPv6 addresses available, you can use the `prefer_ipv6` option to bind to the IPv6 address:

```
# 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

  device = eth0
  port = 4242
  prefer_ipv6 = True
```

To use the TCP Server Interface over Yggdrasil, you can simply specify the Yggdrasil `tun` device and a listening port, like so:

```
[[Yggdrasil TCP Server Interface]]
    type = TCPServerInterface
    interface_enabled = yes
    device = tun0
    listen_port = 4343
```

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
```

(continues on next page)

(continued from previous page)

```
    listen_ip = 127.0.0.1
    listen_port = 5001
    i2p_tunneled = yes
```

In almost all cases, it is easier to use the dedicated `I2PInterface`, but for complete control, and using I2P routers running on external systems, this option also exists.

6.5 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.

The TCP interface types can also tolerate intermittency in the IP link layer. This means that Reticulum will gracefully handle IP links that go up and down, and restore connectivity after a failure, once the other end of a TCP interface reappears.

```
# Here's an example of a TCP Client interface. The
# target_host can be a hostname or an IPv4 or IPv6 address.

[[TCP Client Interface]]
  type = TCPClientInterface
  interface_enabled = True
  target_host = 127.0.0.1
  target_port = 4242
```

To use the TCP Client Interface over Yggdrasil, simply specify the target Yggdrasil IPv6 address and port, like so:

```
[[Yggdrasil TCP Client Interface]]
    type = TCPClientInterface
    interface_enabled = yes
    target_host = 201:5d78:af73:5caf:a4de:a79f:3278:71e5
    target_port = 4343
```

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.

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
```

6.6 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.

Warning: 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 even easier to use.

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

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

  listen_ip = 0.0.0.0
  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.
```

(continues on next page)

(continued from previous page)

```
  # 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
```

6.7 RNode LoRa Interface

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

Warning: Radio frequency spectrum is a legally controlled resource, and legislation varies widely around the world. It is your responsibility to be aware of any relevant regulation for your location, and to make decisions accordingly.

```
# 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

  # It is also possible to use BLE devices
  # instead of wired serial ports. The
  # target RNode must be paired with the
  # host device before connecting. BLE
  # devices can be connected by name,
  # BLE MAC address or by any available.

  # Connect to specific device by name
  # port = ble://RNode 3B87

  # Or by BLE MAC address
  # port = ble://F4:12:73:29:4E:89

  # Or connect to the first available,
  # paired device
```

(continues on next page)

(continued from previous page)

```
  # port = ble://

  # 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 throough 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

  # It is possible to limit the airtime
  # utilisation of an RNode by using the
  # following two configuration options.
  # The short-term limit is applied in a
  # window of approximately 15 seconds,
  # and the long-term limit is enforced
  # over a rolling 60 minute window. Both
  # options are specified in percent.

  # airtime_limit_long = 1.5
  # airtime_limit_short = 33
```

(continued from previous page)

```
  # 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
```

6.13 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`.

  If an interface exceeds its announce cap, it will queue announces for later transmission. Reticulum will always prioritise propagating announces from nearby nodes first. This ensures that the local topology is prioritised, and that slow networks are not overwhelmed by interconnected fast networks.

  Destinations that are rapidly re-announcing will be down-prioritised further. Trying to get "first-in-line" by announce spamming will have the exact opposite effect: Getting moved to the back of the queue every time a new announce from the excessively announcing destination is received.

  This means that it is always beneficial to select a balanced announce rate, and not announce more often than is actually necesarry for your application to function.

- 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*.

6.14 Interface Modes

The optional `mode` setting is available on all interfaces, and allows selecting the high-level behaviour of the interface from a number of modes. These modes affect how Reticulum selects paths in the network, how announces are propagated, how long paths are valid and how paths are discovered.

Configuring modes on interfaces is **not** strictly necessary, but can be useful when building or connecting to more complex networks. If your Reticulum instance is not running a Transport Node, it is rarely useful to configure interface modes, and in such cases interfaces should generally be left in the default mode.

- The default mode is `full`. In this mode, all discovery, meshing and transport functionality is activated.
- The `gateway` mode (or shorthand `gw`) also has all discovery, meshing and transport functionality available, but will additionally try to discover unknown paths on behalf of other nodes residing on the `gateway` interface. If Reticulum receives a path request for an unknown destination, from a node on a `gateway` interface, it will try to discover this path via all other active interfaces, and forward the discovered path to the requestor if one is found.

  If you want to allow other nodes to widely resolve paths or connect to a network via an interface, it might be useful to put it in this mode. By creating a chain of `gateway` interfaces, other nodes will be able to immediately discover paths to any destination along the chain.

  Please note! It is the interface *facing the clients* that must be put into `gateway` mode for this to work, not the interface facing the wider network (for this, the `boundary` mode can be useful, though).

- 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. In addition, path requests from clients on the access point interface will be handled in the same way as the `gateway` interface.

  This mode is useful for creating interfaces that remain quiet, until someone actually starts 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 `roaming` mode should be used on interfaces that are roaming (physically mobile), seen from the perspective of other nodes in the network. As an example, if a vehicle is equipped with an external LoRa interface, and an internal, WiFi-based interface, that serves devices that are moving *with* the vehicle, the external LoRa interface should be configured as `roaming`, and the internal interface can be left in the default mode. With transport enabled, such a setup will allow all internal devices to reach each other, and all other devices that are available on the LoRa side of the network, when they are in range. Devices on the LoRa side of the network will also be able to reach devices internal to the vehicle, when it is in range. Paths via `roaming` interfaces also expire faster.
- The purpose of the `boundary` mode is to specify interfaces that establish connectivity with network segments that are significantly different than the one this node exists on. As an example, if a Reticulum instance is part of

a LoRa-based network, but also has a high-speed connection to a public Transport Node available on the Internet, the interface connecting over the Internet should be set to `boundary` mode.

For a table describing the impact of all modes on announce propagation, please see the *Announce Propagation Rules* section.

6.15 Announce Rate Control

The built-in announce control mechanisms and the default `announce_cap` option described above are sufficient most of the time, but in some cases, especially on fast interfaces, it may be useful to control the target announce rate. Using the `announce_rate_target`, `announce_rate_grace` and `announce_rate_penalty` options, this can be done on a per-interface basis, and moderates the *rate at which received announces are re-broadcasted to other interfaces*.

- The `announce_rate_target` option sets the minimum amount of time, in seconds, that should pass between received announces, for any one destination. As an example, setting this value to `3600` means that announces *received* on this interface will only be re-transmitted and propagated to other interfaces once every hour, no matter how often they are received.
- The optional `announce_rate_grace` defines the number of times a destination can violate the announce rate before the target rate is enforced.
- The optional `announce_rate_penalty` configures an extra amount of time that is added to the normal rate target. As an example, if a penalty of `7200` seconds is defined, once the rate target is enforced, the destination in question will only have its announces propagated every 3 hours, until it lowers its actual announce rate to within the target.

These mechanisms, in conjunction with the `annouce_cap` mechanisms mentioned above means that it is essential to select a balanced announce strategy for your destinations. The more balanced you can make this decision, the easier it will be for your destinations to make it into slower networks that many hops away. Or you can prioritise only reaching high-capacity networks with more frequent announces.

Current statistics and information about announce rates can be viewed using the `rnpath -r` command.

It is important to note that there is no one right or wrong way to set up announce rates. Slower networks will naturally tend towards using less frequent announces to conserve bandwidth, while very fast networks can support applications that need very frequent announces. Reticulum implements these mechanisms to ensure that a large span of network types can seamlessly *co-exist* and interconnect.

6.16 New Destination Rate Limiting

On public interfaces, where anyone may connect and announce new destinations, it can be useful to control the rate at which announces for *new* destinations are processed.

If a large influx of announces for newly created or previously unknown destinations occur within a short amount of time, Reticulum will place these announces on hold, so that announce traffic for known and previously established destinations can continue to be processed without interruptions.

After the burst subsides, and an additional waiting period has passed, the held announces will be released at a slow rate, until the hold queue is cleared. This also means, that should a node decide to connect to a public interface, announce a large amount of bogus destinations, and then disconnect, these destination will never make it into path tables and waste network bandwidth on retransmitted announces.

It's important to note that the ingress control works at the level of *individual sub-interfaces*. As an example, this means that one client on a *TCP Server Interface* cannot disrupt processing of incoming announces for other connected clients on the same *TCP Server Interface*. All other clients on the same interface will still have new announces processed without interruption.

By default, Reticulum will handle this automatically, and ingress announce control will be enabled on interface where it is sensible to do so. It should generally not be neccessary to modify the ingress control configuration, but all the parameters are exposed for configuration if needed.

- The `ingress_control` option tells Reticulum whether or not to enable announce ingress control on the interface. Defaults to `True`.
- The `ic_new_time` option configures how long (in seconds) an interface is considered newly spawned. Defaults to `2*60*60` seconds. This option is useful on publicly accessible interfaces that spawn new sub-interfaces when a new client connects.
- The `ic_burst_freq_new` option sets the maximum announce ingress frequency for newly spawned interfaces. Defaults to `3.5` announces per second.
- The `ic_burst_freq` option sets the maximum announce ingress frequency for other interfaces. Defaults to `12` announces per second.

  If an interface exceeds its burst frequency, incoming announces for unknown destinations will be temporarily held in a queue, and not processed until later.
- The `ic_max_held_announces` option sets the maximum amount of unique announces that will be held in the queue. Any additional unique announces will be dropped. Defaults to `256` announces.
- The `ic_burst_hold` option sets how much time (in seconds) must pass after the burst frequency drops below its threshold, for the announce burst to be considered cleared. Defaults to `60` seconds.
- The `ic_burst_penalty` option sets how much time (in seconds) must pass after the burst is considered cleared, before held announces can start being released from the queue. Defaults to `5*60` seconds.
- The `ic_held_release_interval` option sets how much time (in seconds) must pass between releasing each held announce from the queue. Defaults to `30` seconds.

CHAPTER

SEVEN

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.

7.1 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 needs 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 it's 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.

7.2 Example Scenarios

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

7.2.1 Interconnected LoRa Sites

An organisation wants to provide communication and information services to it's 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.

CHAPTER

EIGHT

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 sys
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.
```

(continues on next page)

(continued from previous page)

```
    destination = RNS.Destination(
        identity,
        RNS.Destination.IN,
        RNS.Destination.SINGLE,
        APP_NAME,
        "minimalsample"
    )

    # We configure the destination 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.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"
        )
```

(continues on next page)

(continued from previous page)

```
        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("")
        sys.exit(0)
```

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 sys
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
```

(continues on next page)

(continued from previous page)

```
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 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)
```

(continues on next page)

(continued from previous page)

```
    # Everything's ready!
    # Let's hand over control to the announce loop
    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 Reticulums Transport
    # system when an announce arrives that matches the
```

(continues on next page)

(continued from previous page)

```
    # configured aspect filter. Filters must be specific,
    # and cannot use wildcards.
    def received_announce(self, destination_hash, announced_identity, app_data):
        RNS.log(
            "Received an announce from "+
            RNS.prettyhexrep(destination_hash)
        )

        if app_data:
            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("")
        sys.exit(0)
```

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

(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("")
        sys.exit(0)
```

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 sys
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 #########################################
##########################################################
```

(continues on next page)

(continued from previous page)

```
# This initialisation is executed when the users chooses
# 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 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.
    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
```

(continues on next page)

(continued from previous page)

```
    # 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))


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:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2
        if len(destination_hexhash) != dest_len:
            raise ValueError(
```

(continues on next page)

(continued from previous page)

```
                "Destination length is invalid, must be {hex} hexadecimal characters (
↪{byte} bytes).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash)
    except Exception as e:
        RNS.log("Invalid destination entered. Check your input!")
        RNS.log(str(e)+"\n")
        sys.exit(0)

    # 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,
```

(continues on next page)

(continued from previous page)

```
                RNS.Destination.OUT,
                RNS.Destination.SINGLE,
                APP_NAME,
                "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.log("Hit enter to manually retry once an announce is received.")
            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)
```

(continues on next page)

(continued from previous page)

```
            rttstring = str(rtt)+" milliseconds"

        reception_stats = ""
        if reticulum.is_connected_to_shared_instance:
            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"
        )
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
        sys.exit(0)
```

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,
```

(continues on next page)

(continued from previous page)

```
        "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)

    # 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)
```

(continues on next page)

(continued from previous page)

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


##########################################################
#### 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:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2
        if len(destination_hexhash) != dest_len:
            raise ValueError(
                "Destination length is invalid, must be {hex} hexadecimal characters (
↪{byte} bytes).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        sys.exit(0)

    # 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,
```

(continues on next page)

(continued from previous page)

```
        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
    # 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
```

(continues on next page)

(continued from previous page)

```
                    )

        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
    # 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")

    time.sleep(1.5)
    sys.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")
```

(continues on next page)

(continued from previous page)

```
        parser.add_argument(
            "-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("")
        sys.exit(0)
```

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 intiator 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

# 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!
```

(continues on next page)

(continued from previous page)

```
    # 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)")

    # 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(link, 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")
```

(continues on next page)

(continued from previous page)

```
    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
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:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2
        if len(destination_hexhash) != dest_len:
            raise ValueError(
                "Destination length is invalid, must be {hex} hexadecimal characters (
↪{byte} bytes).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        sys.exit(0)

    # 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)
```

(continues on next page)

(continued from previous page)

```
        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"
    )

    # 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
```

(continues on next page)

(continued from previous page)

```
                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
    # 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")

    time.sleep(1.5)
    sys.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)
```

(continues on next page)

(continued from previous page)

```
    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",
            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("")
```

(continues on next page)

(continued from previous page)

```
            else:
                client(args.destination, configarg)

    except KeyboardInterrupt:
        print("")
        sys.exit(0)
```

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 sendig 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, link_id, remote_identity, requested_
↪at):
    RNS.log("Generating response to request "+RNS.prettyhexrep(request_id)+" on link
↪"+RNS.prettyhexrep(link_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
```

(continues on next page)

(continued from previous page)

```
    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"
    )

    # 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
```

(continues on next page)

(continued from previous page)

```
# 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")


##########################################################
#### 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:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2
        if len(destination_hexhash) != dest_len:
            raise ValueError(
                "Destination length is invalid, must be {hex} hexadecimal characters (
↪{byte} bytes).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        sys.exit(0)

    # 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)
```

(continues on next page)

(continued from previous page)

```
    # 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
    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
                )
```

(continues on next page)

(continued from previous page)

```
        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.")


# 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")

    time.sleep(1.5)
    sys.exit(0)


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

# This part of the program runs at startup,
# and parses input of from the user, and then
```

(continues on next page)

(continued from previous page)

```
# 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",
            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("")
        sys.exit(0)
```

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

8.8 Channel

The *Channel* example explores using a `Channel` to send structured data between peers of a `Link`.

```
##########################################################
# This RNS example demonstrates how to set up a link to  #
# a destination, and pass structured messages over it    #
# using a channel.                                       #
##########################################################

import os
import sys
import time
import argparse
from datetime import datetime

import RNS
from RNS.vendor import 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"

##########################################################
#### Shared Objects ######################################
##########################################################

# Channel data must be structured in a subclass of
# MessageBase. This ensures that the channel will be able
# to serialize and deserialize the object and multiplex it
# with other objects. Both ends of a link will need the
# same object definitions to be able to communicate over
# a channel.
#
# Note: The objects we wish to use over the channel must
# be registered with the channel, and each link has a
# different channel instance. See the client_connected
# and link_established functions in this example to see
# how message types are registered.

# Let's make a simple message class called StringMessage
# that will convey a string with a timestamp.

class StringMessage(RNS.MessageBase):
    # The MSGTYPE class variable needs to be assigned a
    # 2 byte integer value. This identifier allows the
    # channel to look up your message's constructor when a
    # message arrives over the channel.
    #
    # MSGTYPE must be unique across all message types we
    # register with the channel. MSGTYPEs >= 0xf000 are
```

(continues on next page)

(continued from previous page)

```
    # reserved for the system.
    MSGTYPE = 0x0101

    # The constructor of our object must be callable with
    # no arguments. We can have parameters, but they must
    # have a default assignment.
    #
    # This is needed so the channel can create an empty
    # version of our message into which the incoming
    # message can be unpacked.
    def __init__(self, data=None):
        self.data = data
        self.timestamp = datetime.now()

    # Finally, our message needs to implement functions
    # the channel can call to pack and unpack our message
    # to/from the raw packet payload. We'll use the
    # umsgpack package bundled with RNS. We could also use
    # the struct package bundled with Python if we wanted
    # more control over the structure of the packed bytes.
    #
    # Also note that packed message objects must fit
    # entirely in one packet. The number of bytes
    # available for message payloads can be queried from
    # the channel using the Channel.MDU property. The
    # channel MDU is slightly less than the link MDU due
    # to encoding the message header.

    # The pack function encodes the message contents into
    # a byte stream.
    def pack(self) -> bytes:
        return umsgpack.packb((self.data, self.timestamp))

    # And the unpack function decodes a byte stream into
    # the message contents.
    def unpack(self, raw):
        self.data, self.timestamp = umsgpack.unpackb(raw)


###########################################################
#### 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)
```

(continues on next page)

(continued from previous page)

```
    # 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,
        "channelexample"
    )

    # 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(
        "Channel 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
    latest_client_link = link

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

    # Register message types and add callback to channel
```

(continues on next page)

(continued from previous page)

```
    channel = link.get_channel()
    channel.register_message_type(StringMessage)
    channel.add_message_handler(server_message_received)

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

def server_message_received(message):
    """
    A message handler
    @param message: An instance of a subclass of MessageBase
    @return: True if message was handled
    """
    global latest_client_link
    # When a message is received over any active link,
    # the replies will all be directed to the last client
    # that connected.

    # In a message handler, any deserializable message
    # that arrives over the link's channel will be passed
    # to all message handlers, unless a preceding handler indicates it
    # has handled the message.
    #
    #
    if isinstance(message, StringMessage):
        RNS.log("Received data on the link: " + message.data + " (message created at " +␣
↪str(message.timestamp) + ")")

        reply_message = StringMessage("I received \""+message.data+"\" over the link")
        latest_client_link.get_channel().send(reply_message)

        # Incoming messages are sent to each message
        # handler added to the channel, in the order they
        # were added.
        # If any message handler returns True, the message
        # is considered handled and any subsequent
        # handlers are skipped.
        return True


###########################################################
#### 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
```

(continues on next page)

(continued from previous page)

```
    try:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2
        if len(destination_hexhash) != dest_len:
            raise ValueError(
                "Destination length is invalid, must be {hex} hexadecimal characters (
↪{byte} bytes).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        sys.exit(0)

    # 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,
        "channelexample"
    )

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

    # 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()
```

(continues on next page)

(continued from previous page)

```
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 != "":
                message = StringMessage(text)
                packed_size = len(message.pack())
                channel = server_link.get_channel()
                if channel.is_ready_to_send():
                    if packed_size <= channel.mdu:
                        channel.send(message)
                    else:
                        RNS.log(
                            "Cannot send this packet, the data size of "+
                            str(packed_size)+" bytes exceeds the link packet MDU of "+
                            str(channel.MDU)+" bytes",
                            RNS.LOG_ERROR
                        )
                else:
                    RNS.log("Channel is not ready to send, please wait for " +
                            "pending messages to complete.", 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
    # instance for later use
    global server_link
    server_link = link

    # Register messages and add handler to channel
    channel = link.get_channel()
```

(continues on next page)

(continued from previous page)

```
    channel.register_message_type(StringMessage)
    channel.add_message_handler(client_message_received)

    # 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")

    time.sleep(1.5)
    sys.exit(0)

# When a packet is received over the channel, we
# simply print out the data.
def client_message_received(message):
    if isinstance(message, StringMessage):
        RNS.log("Received data on the link: " + message.data + " (message created at " +␣
↪str(message.timestamp) + ")")
        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 channel example")

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

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

(continues on next page)

(continued from previous page)

```
            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("")
        sys.exit(0)
```

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

8.9 Buffer

The *Buffer* example explores using buffered readers and writers to send binary data between peers of a `Link`.

```
##########################################################
# This RNS example demonstrates how to set up a link to  #
# a destination, and pass binary data over it using a    #
# channel buffer.                                        #
##########################################################
from __future__ import annotations
import os
import sys
import time
import argparse
from datetime import datetime
```

(continues on next page)

(continued from previous page)

```
import RNS
from RNS.vendor import 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"


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

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

# A reference to the latest buffer object
latest_buffer = 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 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,
        "bufferexample"
    )

    # 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(
```

(continues on next page)

(continued from previous page)

```
        "Link buffer 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, latest_buffer
    latest_client_link = link

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

    # If a new connection is received, the old reader
    # needs to be disconnected.
    if latest_buffer:
        latest_buffer.close()


    # Create buffer objects.
    #   The stream_id parameter to these functions is
    #   a bit like a file descriptor, except that it
    #   is unique to the *receiver*.
    #
    #   In this example, both the reader and the writer
    #   use stream_id = 0, but there are actually two
    #   separate unidirectional streams flowing in
    #   opposite directions.
    #
    channel = link.get_channel()
    latest_buffer = RNS.Buffer.create_bidirectional_buffer(0, 0, channel, server_buffer_
↪ready)

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

def server_buffer_ready(ready_bytes: int):
    """
    Callback from buffer when buffer has data available
```

(continues on next page)

(continued from previous page)

```
    :param ready_bytes: The number of bytes ready to read
    """
    global latest_buffer

    data = latest_buffer.read(ready_bytes)
    data = data.decode("utf-8")

    RNS.log("Received data over the buffer: " + data)

    reply_message = "I received \""+data+"\" over the buffer"
    reply_message = reply_message.encode("utf-8")
    latest_buffer.write(reply_message)
    latest_buffer.flush()




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

# A reference to the server link
server_link = None

# A reference to the buffer object, needed to share the
# object from the link connected callback to the client
# loop.
buffer = 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:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2
        if len(destination_hexhash) != dest_len:
            raise ValueError(
                "Destination length is invalid, must be {hex} hexadecimal characters (
↪{byte} bytes).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        sys.exit(0)

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

    # Check if we know a path to the destination
```

(continues on next page)

(continued from previous page)

```
    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,
        "bufferexample"
    )

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

    # 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()
```

(continues on next page)

(continued from previous page)

```
            else:
                # Otherwise, encode the text and write it to the buffer.
                text = text.encode("utf-8")
                buffer.write(text)
                # Flush the buffer to force the data to be sent.
                buffer.flush()


        except Exception as e:
            RNS.log("Error while sending data over the link buffer: "+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
    # instance for later use
    global server_link, buffer
    server_link = link

    # Create buffer, see server_client_connected() for
    # more detail about setting up the buffer.
    channel = link.get_channel()
    buffer = RNS.Buffer.create_bidirectional_buffer(0, 0, channel, client_buffer_ready)

    # 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")

    time.sleep(1.5)
    sys.exit(0)

# When the buffer has new data, read it and write it to the terminal.
def client_buffer_ready(ready_bytes: int):
    global buffer
    data = buffer.read(ready_bytes)
    RNS.log("Received data over the link buffer: " + data.decode("utf-8"))
    print("> ", end=" ")
    sys.stdout.flush()
```

(continues on next page)

(continued from previous page)

```
##########################################################
#### 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 buffer example")

        parser.add_argument(
            "-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:
```

(continues on next page)

(continued from previous page)

```
        print("")
        sys.exit(0)
```

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

8.10 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      #
# 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
```

(continues on next page)

(continued from previous page)

```
# 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"
    )

    # 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 "."
```

(continues on next page)

(continued from previous page)

```
    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)
            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
```

(continues on next page)

(continued from previous page)

```
    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
        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
```

(continues on next page)

(continued from previous page)

```
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:
        dest_len = (RNS.Reticulum.TRUNCATED_HASHLENGTH//8)*2
        if len(destination_hexhash) != dest_len:
            raise ValueError(
                "Destination length is invalid, must be {hex} hexadecimal characters (
↪{byte} bytes).".format(hex=dest_len, byte=dest_len//2)
            )

        destination_hash = bytes.fromhex(destination_hexhash)
    except:
        RNS.log("Invalid destination entered. Check your input!\n")
        sys.exit(0)

    # 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...")
```

(continues on next page)

(continued from previous page)

```
    # 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)

    # 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("")
```

(continues on next page)

(continued from previous page)

```
    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
            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()
```

(continues on next page)

(continued from previous page)

```
            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("")
            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.")
```

(continues on next page)

(continued from previous page)

```
            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:
            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...")
```

(continues on next page)

(continued from previous page)

```
    # And set up a small job to check for
    # a potential timeout in receiving the
    # file list
    thread = threading.Thread(target=filelist_timeout_job, daemon=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")
        sys.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")
    else:
        RNS.log("Link closed, exiting now")

    time.sleep(1.5)
    sys.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):
```

(continues on next page)

(continued from previous page)

```
    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
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__":
```

(continues on next page)

(continued from previous page)

```
    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,
            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("")
        sys.exit(0)
```